LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-527

ASPECTS OF A PARALLEL-ARCHITECTURE SIMULATOR

Eric A. Brewer

February 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Feb. 1992	

4. TITLE AND SUBTITLE

Aspects of A ParallelArchitecture Simulator

5. FUNDING NUMBERS

6. AUTHOR(S)

Eric. A. Brewer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

MIT/LCS/TR 527

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

N00014-89-J

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

14. SUBJECT TERMS

augmentation, execution-driven simulation, multiprocessor simulation, profiling

15. NUMBER OF PAGES

81

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Aspects of a Parallel-Architecture Simulator

by

Eric Allen Brewer

Technical Report MIT/LCS/TR-527

January 1992

This work was supported in part by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and by an equipment grant from Digital Equipment Corporation. Eric A. Brewer was supported by an Office of Naval Research Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Aspects of a Parallel-Architecture Simulator

by

Eric Allen Brewer

Technical Report MIT/LCS/TR-527

Abstract

This thesis discusses the use of code augmentation in PROTEUS, a high-performance parallel-architecture simulator. PROTEUS multiplexes a single processor among the various activities in a simulated parallel machine to provide accurate information about the timing and behavior of an application and the underlying simulated architecture. PROTEUS is fast, accurate, and flexible: it is one to two orders of magnitude faster than comparable simulators, it can reproduce results from real multiprocessors, and it is easily configured to simulate a wide range of MIMD architectures.

Traditional multiprocessor simulators simulate the machine cycle by cycle, interpreting each instruction. The high overhead of instruction interpretation makes these simulators too slow for research in the area of parallel systems. PROTEUS simulates most instructions via a combination of direct execution and code augmentation, which reduces the simulation overhead for these instructions by roughly a factor of one hundred.

In addition to performance, code augmentation offers several other benefits for multiprocessor simulators, such as nonintrusive profiling and stack overflow detection. Primary among these benefits is precise control of the cost in machine cycles of a piece of code. The ability to assign code a cost of zero cycles allows users to generate nonintrusive monitoring and debugging code. Because the monitoring code costs zero cycles, it has no effect on the timing of the simulation. Thus, users can add arbitrary monitoring and debugging code without interfering with the simulation. The ability to assign costs also allows users to tune costs to match a particular architecture, thus increasing the accuracy of the simulation.

Empirical evidence reveals that the performance improvement of augmentation over instruction interpretation is about a factor of one hundred. PROTEUS as a whole outperforms comparable simulators by one to two orders of magnitude. Further experiments reveal that despite its remarkable speed, PROTEUS produces reliable results. In particular, PROTEUS has accurately reproduced published results from several real multiprocessors.

Keywords: augmentation, execution-driven simulation, multiprocessor simulation, profiling

This report is a minor revision of a Master's thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on January 22, 1992. The thesis was supervised by Professor William E. Weihl.

Acknowledgments

I would like to thank Bill Weihl and Barbara Liskov for supervising the project and clarifying some important issues. Barbara was particularly helpful at the beginning, when Chris and I were relatively unfocused and perhaps overly anxious to implement *something* before any of us completely understood the issues. Bill led us to a more modular view of PROTEUS that resulted in well-defined, easily replaced modules; the modularity should help others exploit PROTEUS for their own research.

Of course, most important to this work was Chris Dellarocas. He was always enjoyable to work with, and his distinctly Greek view on life was regularly enlightening. He deserves the success that I know he will achieve.

Many people in Bill Weihl's Parallel-Software Group were helpful both as users and as critics in general. Adrian Colbrook, Anthony Joseph, and Wilson Hsieh were particularly helpful and put up with some terrible bugs. Anthony has a natural ability to encounter the most wicked, hard-to-fix bugs; it's an ability both respected and hated by Chris and me. Paul Wang, Carl Waldspurger, Sanjay Ghemawat, Thanos Papadimitriou, Sharon Perl and Alkarim Allarakhia also deserve credit. I'm sure none of them would inflict similar code on us.

Anant Agarwal and his Alewife group provided insight into networks and caching, and in general gave us an architectural perspective on PROTEUS. David Chaiken, John Kubiatowicz, and Beng-Hong Lim were exceptionally helpful.

Finally, I would like to thank my fiancé, Lisa, for her love and support. She also drew all of the figures that are not simulation graphs.

Eric Allen Brewer

Contents

Acknowledgments 5

1 Introduction 15

1.1 Thesis Organization 16

1.2 PROTEUS Overview 16

1.2.1 Design Goals 17

1.2.2 Architectural Model 18

1.2.3 Running Simulations 21

2 The Goals of Augmentation 23

2.1 Introduction 23

2.2 The Need for High Performance 23

2.3 The Need for Accuracy 26

2.4 Conclusion 28

3 Overview of Augmentation 29

3.1 What is Augmentation? 29

3.2 Augmentation in PROTEUS 31

3.3 Producing Augmented Code 33

3.4 The Flexibility of Augmentation 35

3.5 Fundamental Assumptions 37

3.6 Alternatives to Augmentation 39

3.7 Conclusion 40

4 Detailed Design 43

4.1 The **augment** Program . . . 43
4.2 Basic-Block Boundaries . . . 46
4.3 The Cost of a Block . . . 47
4.4 Counting Cycles and the Quantum Check . . . 47
4.5 The Stack Overflow Check . . . 50
4.6 Profiling . . . 51
4.7 Cycle-Counting Libraries . . . 53
4.8 Turning Counting On and Off . . . 54
4.9 Explicit Control of The Cycle Counter . . . 55
4.10 Optimizations . . . 57
4.11 Conclusion . . . 59

5 Experiments 61

5.1 Measuring the Overhead . . . 62
5.2 *Verifying* the Cycle Counts . . . 63
5.3 Validating the Simulator . . . 64

6 Related Work 69

6.1 Aspen . . . 69
6.2 CARE . . . 70
6.3 Accounting for Time by Hand . . . 70
6.4 Threads . . . 71
6.5 RPPT . . . 71
6.6 Tango . . . 71
6.7 Other Uses of Augmentation . . . 72
6.7.1 Profiling . . . 72
6.7.2 Tracing . . . 73
6.8 Conclusion . . . 73

7 Conclusions 75

Bibliography 79

List of Figures

1-1 The basic architecture model. 19

1-2 The steps involved in building and running a simulation. 22

2-1 Graph involving data from 30 simulations. 25

3-1 Line Counts Generated Through Augmentation. 30

3-2 Clarifying basic blocks. 30

3-3 The process of building a simulator. 34

4-1 A procedure and its *basic-block graph*. 45

4-2 Code generated for cycle counting. 48

4-3 Assembly macro expansion that uses register **$1**. 48

4-4 Code for the stack overflow check. 51

4-5 Procedure-level profile of PROTEUS running eight-queens. 58

5-1 Spinlocks with a small critical section simulated by PROTEUS. 66

5-2 Spinlocks with small critical section on the Sequent Symmetry 67

List of Tables

2.1 The conflict between accuracy and performance. 27

5.1 The overhead of **augment**. 62

5.2 Verifying the cycle counting of **augment**. 63

Chapter 1

Introduction

The PROTEUS system was developed by myself and Chris Dellarocas as a testbed for studying parallel languages and runtime systems, and led to both of our master's theses. Most of the design was done jointly, and all major design decisions involved substantial discussion between the two of us. This thesis covers a small part of the PROTEUS simulation system; it does not even cover all of the components for which the author was responsible.

The author was primarily responsible for the following components:

- The design and implementation of the augmentation program, which is the focus of this thesis.
- The design and implementation of the configuration program (and the associated configuration language), which allows users to specify the simulated architecture. The configuration program is discussed in the PROTEUS user documentation [BD91].
- The design and implementation of the facilities for data collection.
- The design and implementation of the facilities for graphical presentation of simulation data, including the graph-specification language. The data collection and display tools are discussed in the PROTEUS user documentation.
- Much of the support for debugging, including nonintrusive monitoring and debugging code, memory overwrite detection, and stack overflow detection. These aspects are also covered in the user documentation.

This thesis presents a thorough examination of the use of direct execution and code augmentation in a multiprocessor simulation system. Direct execution means that some of the simulated instructions are actually executed on the host workstation; code augmentation is the automated addition of new code to a program to change its behavior in a very specific way. These two techniques are combined in PROTEUS to provide very high-performance simulation of a wide range of multiprocessors, including both shared-memory and message-passing MIMD architectures. The design, implementation, and evaluation of the use of these techniques in PROTEUS is the focus of this thesis.

1.1 Thesis Organization

This thesis is divided into seven chapters. After the general overview of PROTEUS provided by the rest of this chapter, Chapter 2 examines the goals of the project that led to the use of augmentation. The third chapter presents a broad overview of augmentation, followed by a detailed discussion of the design and implementation in Chapter 4. The fifth chapter presents the results of several experiments that investigate the performance and correctness of both augmentation and PROTEUS as a whole. This is followed in Chapter 6 by a discussion of related work. Finally, Chapter 7 draws conclusions regarding the success of augmentation with respect to the goals defined in the second chapter.

1.2 PROTEUS Overview

PROTEUS is not actually a simulator; rather, it is an simulation engine that combines with architecture-specific modules and user applications to create a simulator. The resulting executable provides high-performance simulation of the user's application on the target architecture. This section presents a brief overview of PROTEUS, including the design goals, the basic multiprocessor model, and the steps involved in building and using PROTEUS simulators.

PROTEUS simulates the events that take place in a parallel machine at the level of individual machine instructions, bus or network accesses, interrupt requests, etc. However, the user can write a parallel program for PROTEUS using a simple superset of the C programming language and a set of supporting simulator calls. Parts of the user program that do not require inter-

action with other processors are written in standard C and translated by the C compiler into instructions for the host workstation. Nonlocal interactions in user programs are performed by the supported simulator calls, which roughly correspond to the machine instructions that perform nonlocal interactions in real parallel machines.

1.2.1 Design Goals

In designing PROTEUS, we had five major design goals:

Speed: We were well aware that current simulators had limited usefulness for parallel-systems research because of their poor performance. The challenge was to remove the overhead of cycle-by-cycle instruction interpretation without forfeiting so much accuracy that our simulation results would be suspect.

Versatility/Modularity: PROTEUS must be versatile so that our group can investigate portability and general language and runtime-system mechanisms. Designing portable systems requires the ability to test ideas on a wide range of architectures, including some that physically may not exist. Similarly, general language and runtime-system mechanisms require testing across a broad range of architectures.

Closely associated with versatility is modularity. By designing PROTEUS as a set of replaceable modules, we can reduce and simplify the work required to extend the range of target architectures. For example, it should be possible to change the simulated network without modifying code related to cache coherence or scheduling.

Tradeoff Performance and Accuracy: After starting the project, we realized the importance of providing a range of accuracy and performance options. This goal grew out of our observation that a simple network module based on Agarwal's analytical network model provided an order-of-magnitude improvement in network-simulation performance, and yet often yielded sufficient accuracy. In general, different applications require accuracy in different areas, and a particular application requires different levels of accuracy at different stages of development. Ideally, users should be able to specify the accuracy requirements and achieve the maximum possible performance consistent with those requirements.

Data Collection and Display: Given that simulations should be run to test specific hypotheses, it is critical that the system provide substantial data collection and display tools. It should be easy to collect and display the data needed to resolve a hypothesis. Furthermore, the graphs should be capable of providing new insight into the application and the architecture.

Support for Debugging: Finally, we were very concerned that simulated applications, as with applications on real multiprocessors, would exhibit intractable bugs due to concurrency. Thus, a key goal is *repeatability*, which is the property that rerunning a simulation, even with extra monitoring code, produces the same results—in particular it repeats any bugs. Repeatability allows users to exploit the traditional (and effective) debugging techniques common in sequential program development. In general, the power of standard sequential debuggers should be extended to apply to simulated parallel applications.

1.2.2 Architectural Model

PROTEUS simulates MIMD multiprocessors in which independent processor nodes are connected via an interconnection medium, as shown in Figure 1-1. The interconnection medium can be either a bus, a direct network such as a k-ary n-cube, or an indirect network such as a butterfly. Each processor node consists of a processor, a network chip, a cache chip, and memory. Conceptually, the processor is a generic sequential processor extended with instructions for network access and cache coherence. The network chip interfaces the processor with the interconnection medium. The cache chip, which is optional, handles cache coherence and works with the network chip for remote memory accesses.

The memory at each node is divided into two sections, a *shared* section that maps to part of a global address space, and a *private* section that is not accessible from the interconnection medium. For distributed-memory machines, the size of the shared section is zero. PROTEUS can simulate hardware cache coherence for global memory and provides primitives for software coherence.

Supported machine organizations fall into two major classes: *bus-based* and *network-based*.

Bus-based machines: Bus-based machines are built around a common bus through which

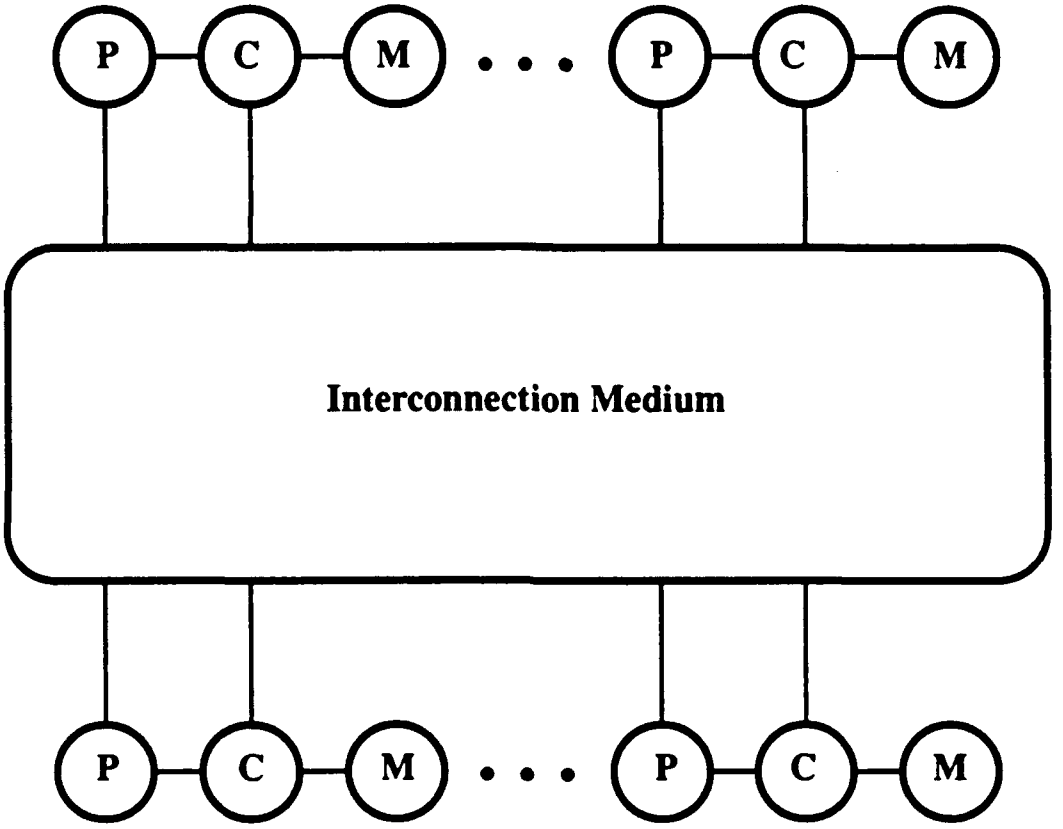

Figure 1-1: The basic architectural model. The interconnection medium can be a bus, a direct network such as a k-ary n-cube, or an indirect network such as a butterfly. The cache chip, which is optional, supports a global address space. For bus-based systems, the number of memory modules may be more than the number of processors, with the extras simply connected to the bus.

processors communicate with shared-memory modules. Interprocessor interrupts (IPIs) are channeled through the common bus or, alternatively, through an optional, independent IPI bus. Processors are independent of shared-memory modules—there may be different numbers of each. Uniform shared-memory access is assumed, that is, access of any memory module from any processor takes the same amount of time (except for delays due to bus contention). Processors are optionally provided with caches whose parameters are specified by the user. Cache coherence is maintained using Goodman's protocol [Goo83], a "snoopy" cache-coherence protocol.

Network-based machines: Machines of this class comprise a number of identical *processing nodes* built around an interconnection network. Each processing node consists of a processor and a shared-memory module. Thus, each memory module is associated with a processor and shared-memory access is non-uniform. Accesses from a processor to its associated memory module are less expensive than accesses to remote memory modules, since requests to the latter must pass through the network. Cache coherence is provided using a version of Chaiken's directory-based cache-coherence protocol [Cha90].

A large variety of direct and indirect network types are supported. *Direct networks* directly connect two processing nodes through point-to-point links. In such networks there is a variable "distance" between two processing nodes, equal to the number of individual point-to-point links that form the path used between those nodes. Indirect networks do not connect any two processors directly, but rather use a number of internal switching stages that automatically route a packet to its destination. In indirect networks, all pairs of processing nodes have the same "distance", which is equal to the number of internal stages plus one.

PROTEUS supports the complete family of k-ary n-cube direct networks with either unidirectional or bidirectional connections. This family includes most direct networks encountered in MIMD machines, such as rings, meshes and hypercubes. The indirect network model can emulate various multistage networks including butterflies and omega networks.

1.2.3 Running Simulations

Figure 1-2 depicts the four steps in the creation and use of a PROTEUS simulator. First, the user specifies the architecture using an X-based configuration tool.

Second, the application- and architecture-specific simulator is compiled and linked into an executable. Augmentation is hidden in this step: the compilation of application procedures includes augmentation of the application assembly code. The compiling and linking step is normally performed either with the configuration program, **config**, or with the **makesim** command. The former allows users to alter the simulated architecture, while the **makesim** command simply rebuilds the simulator for the current architecture. (These are discussed in detail in the user documentation.). In practice, **config** is used only to change the architecture or some of the parameters of the simulation. Most of the time, when the architecture and simulation parameters are stable, the **makesim** command is used to build the simulator, which is an executable called **proteus**.

Next, the user runs the executable to produce screen output and a trace file. The executable can be run either from the **config** program or directly from Unix.

Once **proteus** is executing, the user can interrupt the simulation at any time by pressing `Ctrl-C`. This suspends execution of the simulated program and transfers control to a "snapshot" menu that allows the user to examine the state of the simulated machine at the time of the interrupt. The "snapshot" mode, which is discussed in the user documentation, contains several self-explanatory options and provides information about threads, processors, and memory, and supports application-specific debugging. It also allows the user to continue or abort the simulation.

Finally, PROTEUS includes a sophisticated X-based graph generator, called **stats**, that interprets the trace file and presents the results of the simulation.[1]

[1] All of the graphs in this document were produced by **stats**.

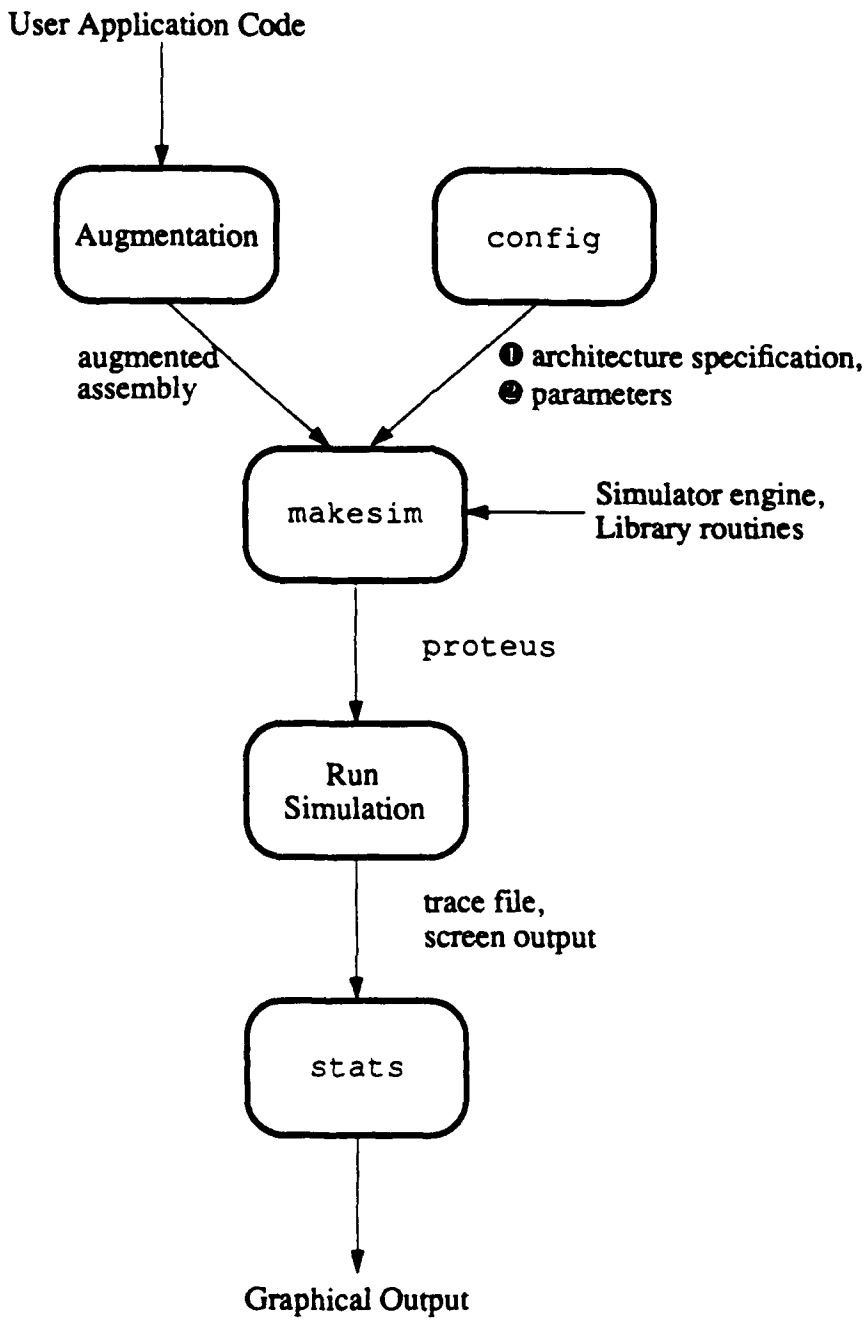

Figure 1-2: The steps involved in building and running a simulation.

Chapter 2

The Goals of Augmentation

2.1 Introduction

Before discussing augmentation in detail, it is useful to examine the goals of the simulator. The decision to use augmentation to account for the costs of local instructions arose from the interaction of two general goals: high performance and simulation accuracy.

2.2 The Need for High Performance

The standard reason for faster simulators is larger simulations: maximum simulation size increases linearly with the speed of the simulator. Like computer performance, simulator performance is always in demand; there will always be a desire for larger simulations and larger collections of simulations. Computer-systems research groups require large simulations, and thus high simulator performance, for at least three reasons.

First, it is important to simulate real applications. Poor performance restricts most simulators to either accurate simulation of tiny programs or large simulations involving many inaccuracies. Unfortunately, toy applications and simulations with questionable assumptions may hide important deficiencies in a language run-time system or in an operating system. Parallel processing is complicated enough that apparently minor aspects of a program can affect performance severely. For example, poor data placement can lead to thrashing in the cache, which increases both access latency and network traffic.

In addition to simulating real applications, a key goal is the investigation of how programs scale. For example, researchers must ensure that synchronization primitives developed for a programming language will be effective for a thousand processors as well as for ten. This of course implies simulating machines with at least hundreds of processors; typical simulator performance precludes such simulations.

The third reason computer-systems groups require large simulations involves the multiple layers of parallel systems. For example, in order to develop a portable parallel language, we need a portable operating system. After we develop the operating system on top of the simulator, we will run applications on top of the simulated operating system. Hence, in order to simulate an application, we must also simulate the operating system. The operating system layer implies larger simulations and an increased need for simulator performance.

Equally important is the ability to run many small simulations. Collections of simulations are useful for comparing a variety of algorithms, evaluating algorithm parameters, and averaging results for more accurate data. Running a large collection of simulations requires a high-performance simulator. In this case, poor simulator performance does not hinder the individual simulations, but rather limits the overall usefulness of the simulator by reducing the quantity and quality of the simulations.

In a study of reader-writer locks, Wilson Hsieh used PROTEUS to compare seven different algorithms on a variety of machine sizes [HW91, HW92]. He also varied the relative percentages of read and write operations. In this three-dimensional space there are six hundred points. Thus, he ran six hundred simulations, not counting the simulations that he had to rerun during development. Obviously, such large collections are feasible only with a high-performance simulator. A typical graph from his work is shown in Figure 2-1. Each point on this graph is the result of one simulation: the thirty points reflect the results of thirty simulations. Other graphs tracked different operation mixes and each required a separate set of thirty simulations.[1]

Adrian Colbrook used the simulator to study the effects of several parameters on two search-tree algorithms. Although he only simulated two algorithms, he studied several variations of one of them, and compared them across several parameters and operation mixes. His simulations

[1] After PROTEUS was established, Anthony Joseph ran a series of over two thousand simulations involving ten-thousand-line fault-tolerance applications. The entire set took only a few days. The existence of PROTEUS changed his methodology; he designed his experiments around the high-performance simulator.

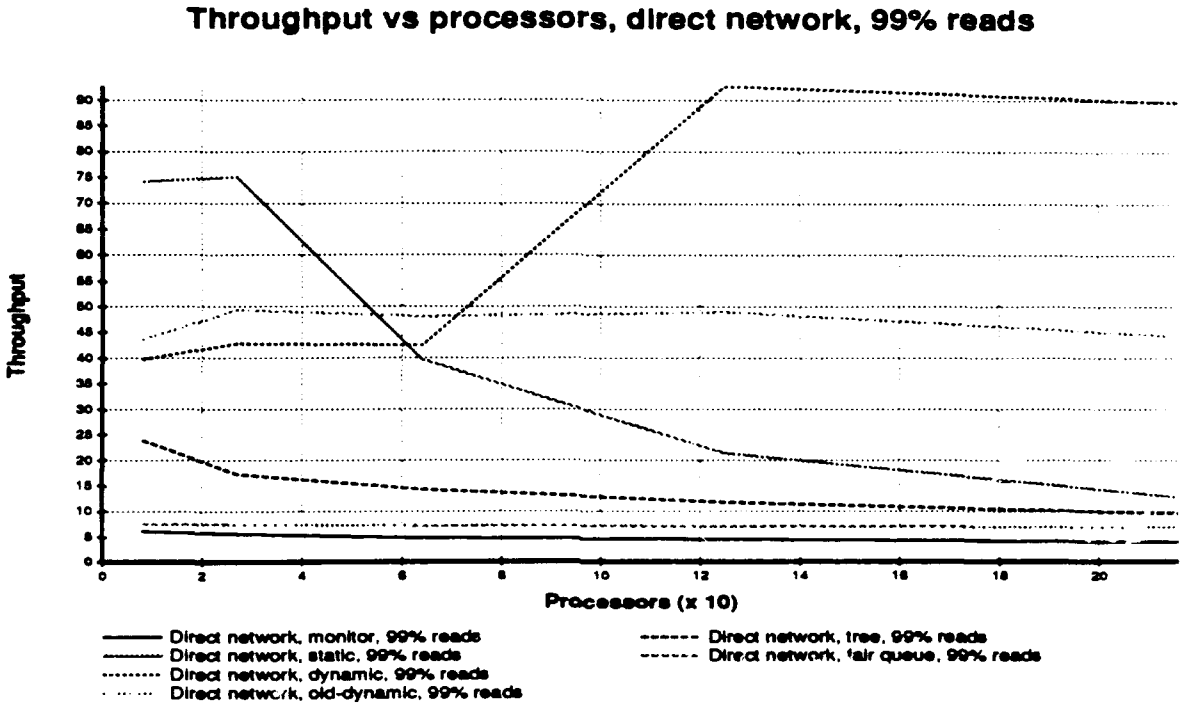

Figure 2-1: A typical graph from Wilson Hsieh's work. In this graph, each of six synchronization algorithms is simulated on 8, 27, 64, 125, and 216 processors, with 99% read operations and 1% write operations.

took less than ten minutes each, so it was never a problem to simulate a new variation or study a new parameter.[2] He estimates that he ran between five hundred and one thousand simulations.[3]

A collection of simulations can also increase the accuracy of the data. Most parallel programs show some form of nondeterminism; execution time, for example, is rarely repeatable (on a real multiprocessor). Slight changes in scheduling or message arrival times can affect greatly the overall performance of the application. In such cases, a single simulation is of little value. More effective is a set of simulations for each data point that generate a mean and standard deviation for the point.[4]

[2] A typical simulation of 50,000 insertion operations takes 2 minutes and 47 seconds.

[3] The conclusions from these simulations appear in "An Algorithm for Concurrent Search Trees" in the 1991 International Conference on Parallel Processing [CBDW91].

[4] Although PROTEUS is a sequential program, events that have the same timestamp can be ordered randomly, which produces nondeterministic behavior very similar to that of real machines. The ordering is determined by a random number seed and is thus deterministic for a given seed; this allows simulations to be repeated exactly for debugging purposes.

Finally, an unexpected advantage of a high-performance simulator is a substantial reduction in development time. Because small simulations take only seconds to execute, application code can be developed incrementally. The fast compile-and-execute cycle greatly reduces the time required to code an application. Many of Adrian Colbrook's five hundred to one thousand simulations tested small parts of the code; such simulations generally take only a few seconds. By the time he ran large simulations, his code was very reliable.

The intended uses of the simulator demand high performance. In particular, the need for large simulations and collections of simulations requires a high-performance simulator. High performance also reduces development time. The need for performance was a driving factor in the decision to use code augmentation.

2.3 The Need for Accuracy

Accuracy of simulation was the second driving factor for using code augmentation. Unfortunately, the goal of accuracy conflicts with the goal of high performance: one can usually speed up a simulation by making it less accurate. The accuracy of multiprocessor simulators varies widely. Hardware designers often simulate a multiprocessor at the level of the VLSI cell. Other simulators interpret each instruction in detail, while still others simulate at a higher level of abstraction. As shown in Table 2.1, each step up in the level of abstraction reduces the accuracy and increases the performance.

Given the conflict between performance and accuracy, it is important to determine the accuracy requirements of the intended uses of the simulator. The requirements differ widely depending on the goals of the application and on its state of development. For example, some applications require very detailed network simulation, while others can get by with a simpler but much faster network module. Furthermore, applications require far less accuracy during development than during data collection. The wide range of accuracy requirements and the conflict between performance and accuracy lead to the following conclusion: it is important to maximize performance by providing only the required accuracy for a particular simulation. Providing more accuracy merely reduces its performance.

Since PROTEUS is intended as a tool for parallel-systems research, it is acceptable to forfeit the ability to simulate machines at the level of the VLSI cell. The harder question is whether

Simulation Level	Cycles per Second
Alewife VLSI Level	2
ASIM Instruction Level with Exact Network	5,000
ASIM Instruction Level with Modeled Network	40,000
PROTEUS with Exact Network and Cache Simulation	120,000
PROTEUS with Exact Network and No Cache	400,000
PROTEUS with Modeled Network and Cache	500,000
PROTEUS with Modeled Network and No Cache	1,000,000

Table 2.1: The conflict between accuracy and performance. This table presents the speed of several simulators on similar hardware. These numbers are only a rough guide; they vary quite a bit depending on the application. Simulations are for machines with 64 processors arranged in an 8x8 bidirectional mesh; machine cycles per second is one sixty-fourth of the "Cycles per Second" number. The exact versions of the network simulate the path of every packet, while the modeled versions calculate arrival time at the target using a model. Only the exact versions reproduce network hot spots. The VLSI number is an estimate by John Kubiatowicz, who designed some of the Alewife hardware. ASIM numbers are from private correspondence with its author, Dan Nussbaum. PROTEUS numbers are for the eight-queens problem.

we must interpret every instruction. In other words, can we forfeit cycle-by-cycle instruction interpretation (and its very high overhead) and still achieve acceptable accuracy for the intended uses of PROTEUS?

One important use of PROTEUS is to test and tune algorithms. In particular, simulator results are used to decide among several algorithms or several variations of an algorithm. Simulator data regarding the speed and scalability of various options must be accurate enough to make the decision reliable. Thus, it is fine if algorithm times are slightly off as long as the relative times are correct and the implications about scaling are correct. If the simulator says one algorithm is twice as slow as another, then we expect the same relative relationship on a real machine. In other words, the graphs generated by PROTEUS should have about the same shape as the graphs from a real machine, although the exact data values can vary somewhat. The validation that PROTEUS meets this goal is discussed in Section 5.3 and is covered in detail in Chris Dellarocas' thesis [Del91].

Unfortunately, many subtle aspects of multiprocessing affect the shape of these graphs. Examples include network congestion, poor shared-memory cache hit rates, and synchronization bottlenecks. The simulator must be able to simulate all of these effects. In particular, PROTEUS

must simulate all accesses to shared memory, must record the size and entry time of every network packet, and must simulate all synchronization primitives.

In general, all *non-local* interactions must be simulated, where non-local essentially means anything that can affect another processor. Examples of *local* interactions include referencing private local memory and executing local instructions such as register-to-register arithmetic. Note that there is no fundamental reason to simulate local instructions exactly. The simulator must reproduce two aspects of local interactions: first, it must reproduce the effects of the interactions, and second, it must measure how many cycles these interactions take. In other words, given a block of local instructions between two non-local interactions, the simulator must reproduce both the effects of the block and the time required to execute the block. If these two requirements are met, other processors cannot tell if PROTEUS interpreted each local instruction in exact detail or if it simply accounted for the time and effects.

Thus, only non-local interactions must be simulated in detail. As discussed in the next chapter, augmentation exploits this property by directly executing local instructions and counting cycles as they execute. Direct execution fulfills the effects requirement, while counting cycles fulfills the timing requirement. The key point is that by providing exactly the accuracy demanded by the intended uses of PROTEUS, we can maximize performance. More accuracy would have little effect on our decisions, but would slow down the simulator significantly.

2.4 Conclusion

The intended uses of the simulator demand high performance, as well as accuracy of all non-local interactions. As discussed in the next chapter, augmentation provides very high performance with sufficient accuracy and allows exact measurement and control of specific costs.

Chapter 3

Overview of Augmentation

The key conclusion from examining the intended uses of the simulator is that although non-local interactions generally require detailed simulation, purely local actions, such as local instructions, can be simulated with less accuracy. The decision to use code augmentation as a fundamental part of the simulator follows from this conclusion. The key idea is to execute local instructions directly and augment the code with cycle-counting instructions to time the code. This chapter presents an overview of augmentation and discusses the flexibility it provides and the assumptions it requires. Finally, it compares direct execution based on augmentation to alternative simulation strategies.

3.1 What is Augmentation?

Augmentation is the process of integrating new code into a code file in order to monitor or add functionality to the code. Augmentation was first described by P. J. Weinberger in 1984 [Wei84]. He used augmentation to add code that counted how many times each source line was executed. Figure 3-1 presents an example. This technique provides very accurate profiling information. Note that augmentation is completely dynamic: although compile-time techniques can often tell very little about dynamic instruction counts, the augmentation counts are determined as the instructions execute and are thus very accurate. Other uses of augmentation are discussed in Section 6.7.

A fundamental concept in the implementation of augmentation is the *basic block*. A basic

Source Code	Counts	Source Code	Counts
`#define N 100000`	1	`max(v, n)`	1
`int x[n];`	1	`int v[];`	1
	1	`{ int i, j;`	1
`main()`	1	`j = 0;`	1
`{ int i;`	1	`for(i=1; i<N; i++) {`	1
`srand(getpid());`	1	`if (v[i] < v[j])`	99,999
`for (i = 0; i<N; i++)`	1	`j = i;`	10
`x[i] = rand();`	100,000	`}`	99,999
`max(x, N);`	1	`return(j);`	1
`}`	1	`}`	0

Figure 3-1: An example profiling output paraphrased from Weinberger's 1984 paper "Cheap Dynamic Instruction Counting" [Wei84]. The code that determines when to exit the loop in **max** occurs not in the **for** line, but at its closing brace. Hence, the closing brace is "executed" 99,999 times.

```
sw  $4, 32($sp)
lw  $14, length
la  $3, string
add $3, $2, 20
```

(a)

```
sw  $4, 32($sp)
lw  $14, length
beq $14, 0, exit
la  $3, string
add $3, $2, 20
```

(b)

```
        sw  $4, 32($sp)
        lw  $14, length
label:  la  $3, string
        add $3, $2, 20
```

(c)

Figure 3-2: Clarifying basic blocks: only the code fragment in (a) qualifies as a single basic block. In (b) the branch instruction (**beq**) implies that the first three instructions could be executed without executing the last two instructions; hence, there are two basic blocks. In (c) the label divides the fragment into two basic blocks, the first ending just above the label.

block is a group of instructions that must be executed as an *atomic* unit. In this context, atomic simply means that the block is indivisible: it is not possible to execute only a proper subset of the instructions in the block. From this definition it is clear that if the first instruction of a basic block is executed then the rest must immediately follow. Figure 3-2 presents code fragments that clarify the definition of a basic block.

Augmentation exploits the atomicity property of basic blocks for higher performance. For example, Weinberger's augmented code counts the number of times each basic block executes instead of the number of times each instruction executes. Because of atomicity, the count for a particular instruction is the same as the count for its basic block. The advantage of counting

basic blocks is that the counting code executes once per block instead of once per instruction. In our simulator, basic blocks usually consist of five to ten instructions, so basic-block counting reduces the augmentation overhead by a factor of five to ten.

3.2 Augmentation in PROTEUS

In PROTEUS, augmentation is used to implement several features. The primary use is counting the cycles required by local instructions. In addition, augmentation is used to improve the safety and accuracy of the simulator and to implement procedure-level profiling. This section provides a high-level overview of each of these uses; a detailed description is presented in the next chapter.

Cycle counting is handled very much like Weinberger's basic-block counting. Instead of counting the executions of the block, the cycle-counting code adds the number of cycles required to execute the block to a global cycle counter that represents the current time. Implicit in the use of cycle counting is the assumption that the instruction set of the simulated processor is similar to the instruction set of the workstation executing the simulation. Section 3.5 discusses this assumption in detail.

By design, augmentation does not always add cycle-counting code; the ability to turn off cycle counting is an important asset of PROTEUS. In particular, users generate nonintrusive monitoring or debugging code by temporarily turning off cycle counting. Since the code is not cycle counted, it does not affect the timing of nonlocal interactions. Without this ability, the debugging code could change the timing, which in turn could cause the relevant symptoms to disappear. Section 4.8 discusses this ability in more detail.

An important safety contribution of augmentation is the addition of code that checks for stack overflow. For single-threaded programs, which include almost all sequential programs, there is only one stack. In a multi-threaded application, however, there may be thousands of stacks. The sheer number of stacks reduces the memory allocated to each stack: although a sequential stack may be several megabytes in size, individual stacks in a multi-threaded program are generally quite small, usually only a few kilobytes. Small stacks greatly increase the chance of stack overflow, which is a very difficult problem to detect and resolve. A stack overflow may overwrite the data of a seemingly unrelated thread, causing errors that are nearly

impossible to resolve because there is no obvious connection between the two threads. The thread that overflowed its stack may not be affected by the error and often appears perfectly normal. Because of the increased likelihood of overflow and the severe problems it can cause, augmentation is used to insert an overflow check into the entry point of every procedure. Although the simulator merely halts when it detects an overflow, it pinpoints the offending procedure and thread. Since the user can adjust the stack size on a per-thread basis, it is usually simple to eliminate the problem.

Augmentation also prevents processors from becoming too far apart in simulated time. Because local instructions are executed directly, a set of local instructions that forms a long or infinite loop causes the simulator to enter the same loop. If a simulated processor loops for a long time it will be far ahead of the rest of the processors in terms of simulated time. This is a problem because an interaction from another processor, such as a message send, may arrive well in the "past" according to the time of the processor in the loop. This implies that the processor has to "undo" all of its work since that time. The infinite loop case is particularly bad, since the simulator enters an infinite loop. To avoid these problems, the simulator limits the maximum time difference between any pair of processors. This is accomplished by establishing a maximum number of cycles, called the *quantum*, that any thread may execute without returning control to the simulator. The quantum does not eliminate an infinite loop, but instead breaks it up into quantum-size chunks. Because the simulator regains control after each chunk, other threads can get useful work done. This is critical because the actions of one of those threads may terminate the "infinite" loop. For example, a thread spinning on a lock will spin until another thread releases the lock; without the quantum, it would spin forever. Because the quantum allows other threads to run, the lock will be released eventually and the spinning thread can obtain the lock.[1] In the case of a truly infinite loop, the quantum allows the user to enter debugging mode by keeping the simulator engine in control. Once in debugging mode, it is easy to determine the existence of an infinite loop.

Finally, PROTEUS uses augmentation to add profiling code to each procedure. Conceptually, this code counts the number of times the procedure executes and tracks the amount of simulated

[1] Actually, spinning is so common that to improve performance the simulator does not really spin. Instead, that processor is marked "spinning" until the desired lock is released.

time spent in the body of the procedure. This procedure-level profiling is based on `prof` [DECd, GKM82], a Unix tool that tracks the amount of time spent in each procedure by periodically sampling the program counter. Unlike `prof`, the augmentation approach is exact. Furthermore, the profiling code tracks simulated time instead of real time, so that code that is not cycle counted is not counted towards the time spent in the procedure.

In summary, augmentation generates code to accomplish four goals. First, it produces cycle-counting code for timing local instruction sequences. The augmentation process also adds code to detect stack overflow, and generates code to ensure that threads execute for at most one quantum before returning control to the simulator. Finally, augmentation produces code for procedure-level profiling. The next section discusses the integration of augmentation with the normal sequence of compiling and linking.

3.3 Producing Augmented Code

In PROTEUS, user code written in a superset of C is compiled, augmented, and then assembled *into machine code*. The process is depicted in Figure 3-3. The **augment** program, which performs the augmentation, converts normal assembly language generated by the C compiler into augmented assembly language that is then assembled into machine language.

The first step converts the superset of C into standard C. The primary difference between the superset and standard C is the ability to declare variables as "shared". This status implies that the declared variable is to reside in globally shared memory. The simulator must know this in order to simulate the cache and network correctly during a shared-variable access. A variable that is not shared resides in private local memory: it can be accessed only by the local processor. Stacks and code generally reside in local memory. The preprocessor that converts the superset into standard C is called `catoc` and is discussed in more detail in Chris Dellarocas' thesis [Del91].

In the second step, the standard C compiler generates assembly language for the source program. This forms the input for augmentation.[2]

Next, **augment** reads the assembly language, divides it into basic blocks, and generates the

[2]We are using the MIPS C compiler.

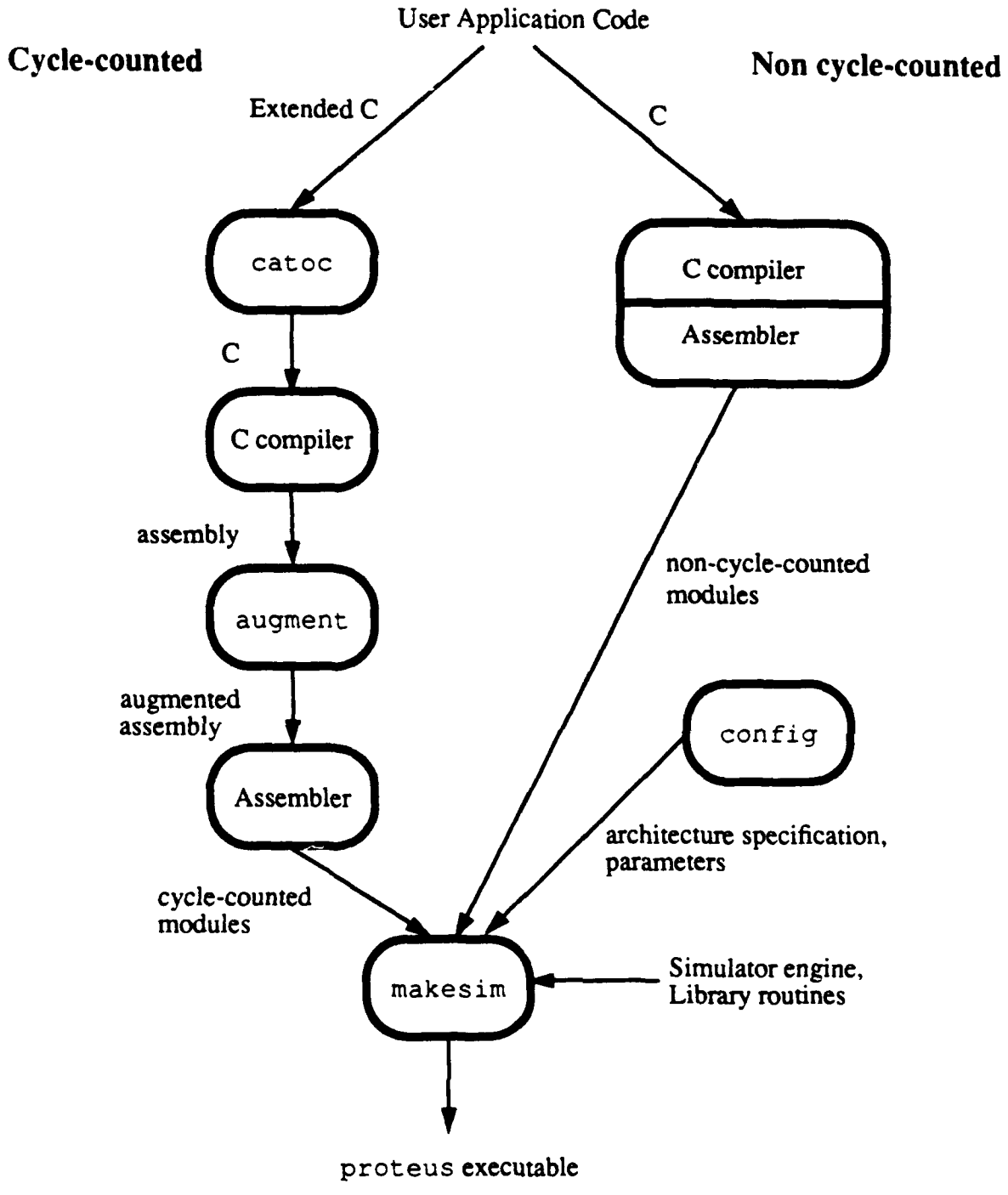

Figure 3-3: Constructing a simulator executable. Cycle-counted user code written in a superset of C is transformed first into normal C and then into assembly language. The code is then augmented and assembled to produce a cycle-counted object module. Non-cycle-counted code is compiled directly into non-cycle-counted modules. The **makesim** program combines user modules, configuration information, the simulator engine, and libraries to produce the **proteus** executable.

augmented assembly language. The next chapter discusses the internal steps of **augment** in detail.

Finally, the augmented assembly language is assembled into an object file and linked with the rest of the PROTEUS engine to form the simulator executable, normally called **proteus**.

An important point about this process is the large amount of work that the simulator passes off to existing tools. The C compiler, the assembler, and the linker are just the normal Unix utilities. Only the preprocessor and **augment** required any development time.

3.4 The Flexibility of Augmentation

Although performance and accuracy were the most important factors in the decision to use augmentation, the flexibility provided by augmentation is an important asset. The flexibility derives from the ability to measure and alter specific costs.

Because both local code and non-local interactions are timed in cycles, it is very easy to measure the time taken between two points: users simply record the cycle time at each point and compute the difference. Many simulation strategies, including instruction interpretation, have this property but many others do not. For example, systems that use fixed costs for procedures cannot resolve times at a finer granularity than procedures. Similarly, systems that use an external timer to measure time are limited by the coarseness of the timer and by the interference of the code added to record the time. Section 3.6 presents an overview of these alternative strategies, and Chapter 6 discusses their use in other simulators.

Profiling is an example of the usefulness of such measurements. A profiling tool such as **prof** [DECd] can resolve performance problems to specific procedures. As discussed in Section 4.6, nonintrusive profiling code can be added during code augmentation.

While profiling provides performance information at the procedure level, it is often useful to measure costs within a procedure. For example, such measurements can reveal synchronization bottlenecks by simply measuring the time across a synchronization primitive. In fact, users of PROTEUS find that the ability to measure point-to-point times reduces the need for procedure-level profiling.

More useful than the ability to measure costs is the ability to alter them. Unlike simulators that interpret instructions, the notion of time is tied to instructions only indirectly. While

most instruction-interpreting simulators fix the cost of an instruction, the use of augmentation separates the functionality of a block of code from the time it takes to execute. Normally, the cost of the block is set to the cost in cycles of the instructions it contains. However, this assignment is easily adjusted. For example, the cost of debugging code, such as a print statement, can be measured and subtracted out. This results in a block that executes the debugging code "for free": the extra code has absolutely no effect on the rest of the simulation. The value of this ability is sometimes priceless: usually the addition of debugging code slightly changes the timing, which may cause the bug to disappear. This problem, called the *probe effect* [Gai86], is fundamental to multiprocessors, but can be eliminated in PROTEUS by ignoring the execution time of the debugging code.

As a second example, consider the code for the operating system our research group is developing. Once it is stabilized there is no need to count its costs every time, since they will not change. Instead, we can measure the costs once and set them directly thereafter. The advantage of this approach is that we can throw out the cycle-counting code; this improves the performance of the simulator without affecting its accuracy.

The ability to alter times simplifies simulating new machines. For example, message-sending overhead on the N-Cube [FJL+88] is much higher than on the J-Machine[3] [D+89], even though both are distributed-memory multiprocessors. Instead of writing message-passing code for each machine, we can just set the costs to match those of the desired machine. Thus with one set of message-passing routines, we can simulate the overhead of many machines.

Finally, altering costs allows us to extend the power of the simulator. An example of this phenomenon is the simulation of moving a stack on a distributed-memory machine with per-word hardware tags. (Such a move is required to migrate an object to another processor.) The algorithm for this is quite simple: traverse the stack and translate all pointers to the corresponding locations in the new address space and simply copy all non-pointers. Our current implementation cannot easily simulate this algorithm because the workstations we use do not have hardware tags.[4] Because the simulator actually has a single address space, we do not

[3]The J-Machine is a research machine being developed at MIT by the Concurrent VLSI Architecture group headed by William Dally.

[4]There are a variety of difficult ways to simulate hardware tags using software tags, but all of them are quite slow.

really have to move the stack: it is sufficient to account for the time it would take to move the stack on a machine with hardware tags. The time required is essentially an affine function of the length of the stack. Thus, instead of actually moving the stack, the code simply pretends to be busy for the right amount of time. The ability to alter costs allows the simulator to simulate stack moving correctly even though the underlying hardware cannot move the stack.

3.5 Fundamental Assumptions

The use of augmentation to simulate local instructions requires some fundamental assumptions that merit further discussion. This section investigates three assumptions, discussing both the problems they resolve and the problems they create. Despite being logically invalid, these assumptions are *reasonable* in the sense that they do not affect the results of simulations significantly. The proof that they are reasonable lies in the ability of PROTEUS to reproduce data generated by real multiprocessors, which is discussed in Section 5.3 and covered in detail in Chris Dellarocas' thesis [Del91].

The first assumption is that every instruction executes in a statically known fixed number of cycles. Because basic blocks are assigned a fixed number of cycles per execution, augmentation must assume that the block requires a fixed number of cycles. The existence of caches makes this assumption false: an instruction requires more cycles if it is not in the cache. Furthermore, if the instruction accesses local memory, the access time depends on the cache. It is essentially impossible to determine statically the number of cycles required by an instruction.

Consequently, PROTEUS assumes that the cache hit rate for code and *local* data is 100%. Fortunately, the hit rate of uniprocessor caches is very high, generally over 95%, so this assumption is probably reasonable. Since this assumption affects only local instructions, it is the hit rate of uniprocessor caches that is relevant, not the hit rate of multiprocessor caches, which is much lower. Finally, the error in the cycle count due to cache misses is minor compared to the effects of the compiler. Changing compilers or even just changing some of the compiler's parameters can affect the number of instructions in a block by more than ten percent. The large effect of the compiler is demonstrated in Section 5.2. The key point is that small errors in the costs of local instructions do not affect the conclusions drawn from simulation results.

The second fundamental assumption is that the local instruction set of the simulated proces-

sor is the same as the local instruction set of the workstation executing the simulation. Because local instructions are executed directly instead of being interpreted, PROTEUS must assume that the local instructions of the workstation correspond to those of the simulated processor. The proliferation of RISC architectures mitigates the effects of this assumption: the costs of most instructions are quite uniform across the range of RISC instruction sets, which includes both our workstations and the multiprocessors that we have investigated. Although most instruction sets for multiprocessors have some instructions that are not present in uniprocessor architectures, the *local* instructions match quite well. Finally, the exact costs of local instructions are relatively unimportant: we are interested in parallel systems behavior, so small differences in local instruction times are second-order effects at best. We expect the workstation instruction set to *represent* the local instructions rather than match them exactly; slight differences in local instruction costs should not affect the conclusions drawn from simulations. Section 4.3 discusses the cost of individual instructions and the overall cost of a block.

The third assumption is that interrupts that arrive in the middle of a block of local instructions do not affect the functionality of the block. Since PROTEUS executes the block directly, it does not reproduce the effects of an interrupt that arrives (in simulated time) during the block. Instead, the interrupt executes after the block has returned control to the simulator. The functionality of the block should not change if the interrupt actually arrived in the middle, since interrupt handlers do not affect the code interrupted other than the delay in execution. Non-local instructions do not place this constraint on interrupts, since they are interpreted. Code that is supposed to be affected by interrupts contains non-local instructions by definition.

Although this section argues that the assumptions required by augmentation are reasonable, it is not clear that simulation data actually match reality given these assumptions. Fortunately, there is direct evidence that these assumptions are reasonable. For example, our simulation of David Chaiken's coherent-cache protocol [Cha90] reproduces all of the problems discussed in his thesis; furthermore, applying the solutions he suggests leads to the same performance problems and benefits that he encountered. Section 5.3 presents a more detailed example, comparing simulation results with published results from a real multiprocessor. The nearly identical behavior is strong evidence that these assumptions are reasonable. In general, any effect that we have expected to see actually has appeared.

More importantly, all unexpected effects have so far proven to be fundamental behavior rather than idiosyncrasies of the simulator. For example, Wilson Hsieh found several fundamental spinlock effects that do not appear in the literature, but are easily supported by analysis. These effects are (probably) not in the literature because they are very subtle on the shared-memory multiprocessors currently in use, which generally have less than sixty-four nodes.

3.6 Alternatives to Augmentation

The simulation of local instructions can be handled in a variety of ways in addition to augmentation. Although some of these have been touched on before, and all are revisited in Chapter 6, it is useful to summarize and contrast these other approaches. There are four approaches besides augmentation for simulating local instructions: (1) interpreting instructions, (2) assigning costs to local code, (3) timing local code using a hardware timer, and (4) counting cycles using a hardware cycle counter.

The most common and most accurate approach is to interpret each instruction. This is extremely accurate since it simulates every memory reference, every register, and all of the condition flags. Unfortunately, it is also very slow. It generally takes hundreds of instructions to interpret a single simulated instruction. According to the author of ASIM, the instruction-level simulator for the Alewife multiprocessor being developed at MIT [CLN90], ASIM executes about two hundred instructions to simulate one instruction. In contrast, the overhead from augmentation in PROTEUS is about a factor of two.[5] The high overhead of instruction interpretation makes it an unacceptable approach given the intended uses of simulator.

At the other extreme of the accuracy versus performance spectrum is simply assigning costs to local code [Riz89]. This approach should be faster than augmentation, since one update to the current time is made for the entire piece of code, instead of one update per executed basic block. Unfortunately this approach can be wildly inaccurate. There may not even be a single cost for a piece of code: it may take ten cycles for one execution and ten thousand

[5]The overhead from augmentation is so small that it is dwarfed by the overhead for simulating the caches and the network, and even by the overhead for context switching. Thus, the speed increase of PROTEUS over ASIM is only a factor of fifteen instead of the implied factor of one hundred. Section 4.10 discusses the relative overhead costs more completely.

for another. This problem can be mitigated somewhat by making the cost dependent on some run-time parameters, but then this approach starts to look like augmentation. Even if there is a single cost, it may not be intuitive. Some apparently simple operations, such as computing a square root, can take thousands of cycles. Finally, it is relatively easy for the assigned costs to become incorrect when the code they represent is changed. This is not a problem in the other approaches, where the cost is recalculated automatically whenever the code changes. All of these problems make the computed times inaccurate, probably inaccurate enough to make the simulation results unreliable.

Closer to augmentation is the use of a hardware timer [Els91, DSNB87]. The use of a timer results in less overhead than augmentation, but is less accurate. This technique depends on a fine-resolution hardware timer. The standard timer for Unix has millisecond resolution, which is far from sufficient on modern workstations. The workstations we use[6] run at more than twenty million instructions per second, or about twenty thousand instructions per millisecond. Thus the Unix timer is accurate to within about ten thousand instructions. This is far too coarse for accurate simulations. A microsecond timer, absent from most machines, would be accurate to within ten instructions. Typical blocks of local code are ten to forty instructions, so ten instructions is significant. The accuracy is decreased further by the probe effect: the measured time includes the execution time of the recording code.

A hardware cycle counter would be faster and just as accurate as augmentation. Unfortunately, such counters are not available yet. Note that there is little difference between a cycle counter and a hardware timer with single-cycle resolution. Given the above options, even if we assume a microsecond timer, augmentation is the logical approach for simulating local code.

3.7 Conclusion

This chapter presented an overview of the functionality provided by augmentation, discussed the flexibility and assumptions of augmentation, and discussed alternative approaches for simulating local code. Augmentation generates code to count cycles, detect stack overflow, limit threads to a quantum of execution time, and record profiling information; the next chapter describes these

[6] We are currently using DECstation 5000s.

tasks in detail. Augmentation also promotes precise control and measurement of costs, which allows nonintrusive debugging and monitoring and separates the cost of an operation from its functionality. Finally, although augmentation requires several fundamental assumptions, these assumptions have little impact on the validity of conclusions drawn from simulations.

Chapter 4

Detailed Design

While previous chapters discuss augmentation at a high level, this chapter presents the detailed design of the augmentation process. The implementations of cycle counting, the quantum, stack overflow detection, and profiling are presented in detail. Additional functions of **augment** are also discussed, including generating code for accessing the current time and automatically linking in cycle-counting libraries. Finally, this chapter looks at some possible optimizations for augmentation.

4.1 The augment Program

The **augment** program is oriented around procedures. Although an input assembly language file may contain many procedures, **augment** treats the file as a collection of independent procedures. The independence simply means that the augmentation of a particular procedure is not affected by the rest of the procedures. A second general rule is that **augment** only adds lines, it does not modify or delete them. This rule is expedient for two reasons. First, it is much easier to ensure that augmentation does not affect the functionality of the application if the source lines pass through untouched. Second, any comments in the source file are propagated through to the output, which makes the augmented code easier to understand.

At the topmost level, **augment** scans for procedures and analyzes each one, building a list of required changes for each procedure. A second pass is then made through the input file. In this pass, lines are copied to the output file until **augment** reaches a *point of augmentation*, which is

a point where code must be added. After the extra code is output, **augment** continues copying the input until the next point of augmentation.

The first step in **augment** is to parse the input file into labels, instructions, registers, and the other elements of the assembly language. This is done using a lexical analyzer generated by **lex** [DECb], and a simple top-down parser. The parser has three modes: data, text, and subroutine. Data mode occurs when the source file contains statements for the data area of the executable. Text mode handles definitions that are outside of procedures, and subroutine mode handles source lines that are part of a procedure. Only subroutine mode leads to any augmentation.

After **augment** locates a procedure, the real work of augmentation begins. The procedure is broken into basic blocks and the *basic-block graph* is generated. Each node in a basic-block graph corresponds to one basic block, and contains information such as the starting and ending line numbers and the number of cycles required to execute the block. The edges of the graph define the possible next blocks to execute. The only case in which a block does not have exactly one next block occurs when a block ends with a conditional branch, in which case there are two outgoing edges. Figure 4-1 shows a procedure and its basic-block graph.

Once the basic-block graph is complete, it is *compacted*. The goal of compaction is to increase the average basic-block size and thus reduce the overhead of augmentation. Compacting the graph eliminates zero-instruction blocks and concatenates adjacent blocks if the second block begins with an unreferenced label. The latter optimization occurs more than one might expect because the compiler generates all of the labels it *might* reference. For example, **return** statements generally produce code that branches to the epilogue; the compiler generates the epilogue label regardless of whether or not the procedure contains any **return** statements.

After all of the basic-block graphs are complete, one for each procedure, the output phase begins. The line-number fields of the basic-block nodes are used to locate each block in the source file. The nodes are stored in order by line number, so it is always clear which block will occur next in the source file. As described above, the output routines alternate between copying the source file and generating the per-node additional code. When **augment** reaches the end of the source file, the process is complete.

```
int fact(x)
int x;
{
   if (x < 2) {
      return 1;
   } else {
      return x*fact(x-1);
   }
}
```

(a) Procedure `fact`

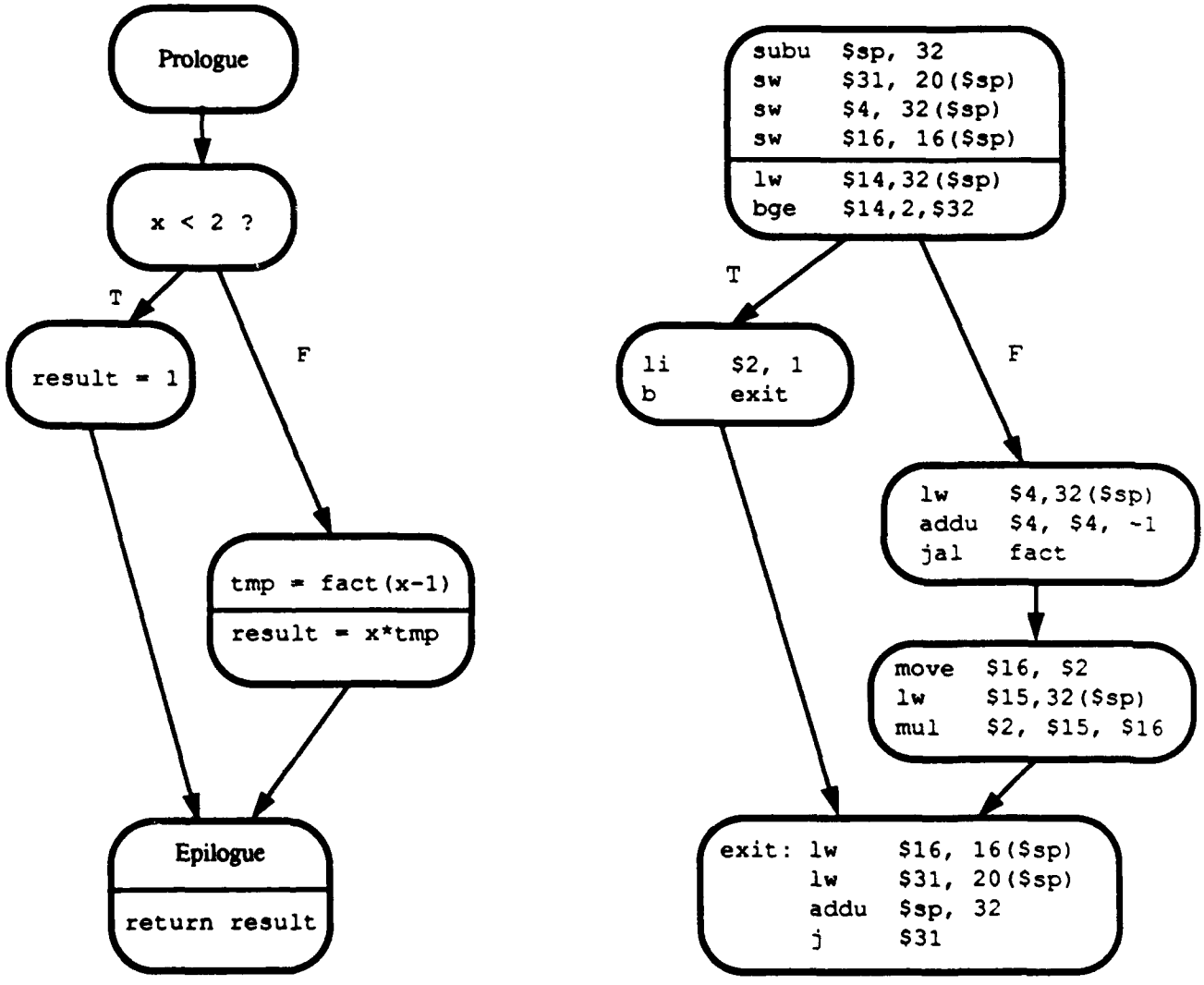

Figure 4-1: A procedure and its basic-block graph. The procedure **fact**, listed in (a), has the basic-block graph shown in (b). The final basic blocks and their graph are shown in (c). Note that the procedure call to **fact** splits one of the nodes in (b) into two nodes in (c), and that the prologue and the first block in (b) have been merged in (c).

4.2 Basic-Block Boundaries

Basic blocks begin and end at four types of statements. When a statement ends one block, the next line implicitly begins the next block. The first of these statements is the branch instruction, which must end a block because the following instruction is not executed if the branch is taken. Thus atomicity requires that the two instructions reside in different blocks.

Likewise, a label starts a new block because statements on opposite sides of the label need not be executed together. If the label is unreferenced, then the two sides form one basic block, since it is impossible to jump into the middle of the block. The compaction phase identifies unreferenced labels and merges the corresponding blocks.[1]

The third statement that delimits basic blocks is the procedure call. Since a procedure call might not return, it can be viewed as a conditional branch; like a branch statement, the call instruction ends a block. Although procedures almost always return, there is a more important reason to end the block: the called procedure may reference the current time, so the time must be correct at the point of the call.

Consider the effects of not ending a block at a procedure call. The code to add the cycles for the block occurs either before or after the call. If it occurs before, then the time required to execute the instructions after the call, but in this block, is counted by the time the call executes. Thus the current time is off by the number of cycles required to execute the part of the block following the call. Likewise, if the cycles are added after the call, then the instructions that reside in this block before the call are not counted by the time the call executes. Again the time is off. The correct solution is to count the instructions up to and including the call instruction before the call executes, and count the instructions after the call after the procedure returns. The easiest way to do this is to begin a new block after a call instruction and ensure that the code for counting cycles precedes the call instruction.

The fourth delimiter follows similar reasoning. A new block begins after the user reads the current time. If the access did not end a block then the time read would be off, just as in the case above. To read the time, users read a dummy global variable, `current_time`; `augment` recognizes the access and translates it into code that computes the current time. This code is

[1] It is not possible to determine if a label is unreferenced during the first pass, since references may occur later in the file. Compaction takes place after the graph is complete, so the status of each label is accurate.

not cycle counted.

4.3 The Cost of a Block

The cost of a block is simply the sum of the costs of its instructions. The basic cost of each instruction is stored in a table and can be adjusted to reflect the costs of the simulated processor. Unfortunately, neither the exact costs of the instructions nor even the exact instructions in the block can be determined from the assembly language.

As discussed in Section 3.5, the actual run-time costs of the instructions depend on the cache; **augment** assumes that the hit rate is constant and uniform. Thus, the extra cost due to cache misses is reduced to a single factor that is slightly greater than one. The cost of a block is the basic cost of its instructions, assuming no cache misses, times the cache-miss overhead factor.

The exact instructions in the block depend on the macro expansions performed by the assembler and the number of pipeline delays that it inserts. Unfortunately, the MIPS assembler is smart enough to use context-specific expansions. For example, multiplication by a constant sometimes expands into shifts and adds and other times expands into a multiply instruction. In the case of several expansions with differing costs, **augment** uses the average cost of the macro.[2]

4.4 Counting Cycles and the Quantum Check

At the beginning of every basic block, code is added to track the cycles required to execute the block. To implement the quantum, the resulting time is checked against the expiration time for this quantum. For better performance, the counter for the current quantum actually counts down to zero instead of up to a maximum. This is faster because a comparison against zero is faster than a comparison against a location in memory.

The simulator engine maintains two global variables that represent the current time. The global variable **cycles_** is set to the quantum value just before the user thread executes. A second global variable, **base_time_**, which is also set by the simulator engine, is used to compute

[2]It uses the expected value, weighing the cost of each expansion by the probability of that expansion. The probabilities were determined by examining a random sample of the macro in question.

```
      lw  $8, cycles_
      sub $8, $8, x
      sw  $8, cycles_
      bgtz $8, L1
      jal SimQuantum
L1:   ...
```

(a)

```
      .noat
      lw  $1, cycles_
      sub $1, $1, x
      sw  $1, cycles_
      bgtz $1, L1
      jal SimQuantum
      .at
L1:   ...
```

(b)

```
      lw  $1, cycles_
      sub $1, $1, x
      sw  $1, cycles_
      bgtz $1, L1
      sw  $31, -4($sp)
      jal SimQuantum
      lw  $31, -4($sp)
L1:   ...
```

(c)

Figure 4-2: Code generated for cycle counting. In (a) register **$8** is available for updating **cycles_**. In (b) the **.noat** and **.at** directives warn the assembler that **$1** is in use. In (c) the return instruction pointer is saved on the stack across the call to **SimQuantum**.

Macro	Expansion
`bgt $11, $12, label`	`sgt $1, $11, $12` `bnz $1, label`

Figure 4-3: Example of the assembler's use of register **$1**: the assembly macro **bgt**. The MIPS R3000 processor does not have a "branch if greater than" instruction. Instead the **bgt** macro implements it with two instructions: the first compares the registers, using register **$1** to hold the result temporarily, and the second performs a conditional branch based on the result.

the current time:

current time = **base_time_** − **cycles_**

Only **cycles_** is modified by the augmented code; **base_time_** can be read but not written. The thread executes until a call to the simulator engine is executed, such as an access to shared memory, or until the value of **cycles_**, which decreases as the thread executes, becomes less than or equal to zero. Figure 4-2a shows the code added for cycle counting and the quantum check.

A key problem in updating **cycles_** is finding an available register. To do this, **augment** tracks which registers are used by the procedure. If one of the registers is not used, then this register is used to update **cycles_**. If all of the registers are in use, then the code uses register **$1**, which is normally reserved for the assembler. Assembly language macros use register **$1** as a temporary register, as shown in Figure 4-3. The register can be used as long as such

macros are avoided. The directives `.noat` and `.at` disable and enable the assembler's use of the register.[3] The assembler generates an error if it needs the register while it is disabled. Figure 4-2b presents this version of the cycle-counting code.

Because not all processors provide a register for temporary storage, implementations on other machines may require more sophisticated techniques to find a temporary register for the cycle-counting code. One possibility is more exact tracking of register usage. The current version simply detects if a register is used anywhere in the procedure. A better algorithm would determine exactly which registers are live at the point of augmentation. Even with exact knowledge of register usage, all of the registers may be in use. One general solution is to save the contents of a register on the stack, thus creating a free register. This approach was avoided on the MIPS because it requires two extra memory accesses compared to the use of register $1.

The quantum expiration creates a second problem. If the quantum has expired, the code calls the procedure **SimQuantum**, which returns control to the simulator. The MIPS processors use a "branch and link" instruction for procedure call: the return instruction pointer is saved in register **$31** and the processor branches to the procedure. The return instruction loads the instruction pointer from register **$31**. The goal of "branch and link" is to avoid saving the instruction pointer on the stack unless it is strictly necessary. For example, "leaf" procedures, those that do not call other procedures, need not save the instruction pointer on the stack, since they can simply leave the instruction pointer in register **$31**. Procedures that call other procedures, however, must move the saved instruction pointer out of register **$31** to prevent subsequent calls from overwriting it. Such procedures usually save register **$31** on the stack for the duration of the procedure.

The problem is that the call to **SimQuantum** must avoid writing over the saved instruction pointer. The simple solution is to save register **$31** on the stack across the call. This is unnecessary if the register has already been saved on the stack. As an optimization, **augment** saves register **$31** only if the procedure has not already saved it on the stack. To implement this, each node in the basic-block graph has a flag indicating the need to save register **$31**. Figure 4-2c includes the code to save register **$31**.

A critical property of the cycle-counting code, including the quantum check, is that it has

[3]The "at" in these directives stands for "assembler temporary".

no effect on the functionality of user code. The register used by this code is not used by the user code. Although the extra code affects the performance of the original code, it does not interfere with the functionality.

4.5 The Stack Overflow Check

While the cycle-counting code affects every basic block, the code for detecting stack overflow is generated only once per procedure. Most compilers allocate stack space during the procedure prologue and release it in the epilogue. Thus finer granularity stack checking would reduce performance without increasing safety. The code for overflow detection is inserted into the prologue *after* the stack frame has been allocated. This ensures that this procedure, but not necessarily its children, will complete without overflowing the stack. The procedure's children must check their own stack usage.

On our workstations, the stack grows downward: the compiler allocates a frame by subtracting the frame size from the stack pointer. A stack overflow is thus equivalent to the stack pointer falling below some minimum address. The global variable **stackmin** stores this address for the current stack; **stackmin** is set by the simulator just before resuming a thread. The overflow detection code simply compares the stack pointer, which has been adjusted to include the current frame, and **stackmin**. If an overflow has occurred, the code calls the procedure **StackOverflow**, which identifies the thread and procedure and halts the simulator. Figure 4-4 presents the overflow detection code.

Procedures that are not augmented normally do not detect overflow. Since users can create non-augmented procedures this is a serious loophole. Fortunately, several factors mitigate the problem. First, procedures can detect stack overflow by calling the routine **CheckStack**, which is similar to the code added by **augment**. Second, if the non-augmented procedure calls an augmented procedure, the latter will catch a stack overflow. Third, non-augmented procedures tend to be small in practice and thus less likely to overflow the stack.

Finally, the simulator has an adjustable *threshold* for stack overflow. For example, if the threshold is 800 bytes, then **stackmin** is set to 800 bytes from the end of the stack. Thus a stack overflow really means that the stack pointer entered the threshold area, and may or may not have overflowed its allocated space. Although this wastes space, it increases safety by allowing

```
                lw   $25, stackmin
                sub  $25, $sp, $25
                bgez $25, L103
                jal  StackOverflow
L103:           ...
```

Figure 4-4: Code added to perform the stack overflow check. This code is inserted after the prologue of every cycle-counted procedure. The simulator engine ensures that the global variable **stackmin** always contains the overflow address for the current stack. Overflow occurs when the stack pointer (**$sp**) falls below **stackmin**. Procedure calling conventions ensure that register $25 is free at procedure entry, and **augment** guarantees that the new label, L103 in the above code, is unique. Finally, note that the procedure **StackOverflow** never returns, since stack overflow is a fatal error.

non-augmented procedures to use stack space up to the threshold value without overflowing the stack. With the threshold at 800 bytes, for example, a non-augmented procedure is guaranteed at least 800 bytes before overflowing the stack. In practice, the threshold is kept small most of the time, usually a few hundred bytes.

Note that users can rule out errors due to undetected stack overflows by increasing the stack size, which can be set from the **config** program. If the errors disappear, then overflow was probably the cause.

Although stack overflow has appeared only rarely, its quick detection has saved countless hours of debugging. Overwrite bugs, such as stack overflow, are by far the hardest to track down. Users can generally locate the area that is being overwritten, but it is very difficult to track down the source lines that overwrite the area. Even if the user can narrow down the time of the overwrite, any of the threads active during that time could be the culprit. The PROTEUS user documentation [BD91] discusses support for resolving overwrite bugs. Although the overflow detection code merely halts the simulator, it greatly simplifies debugging by immediately detecting and locating the problem.

4.6 Profiling

PROTEUS uses augmentation to add code that records profiling information as a procedure executes. The profile reveals both the number of calls for each procedure and the amount of

simulated time spent in the body of each procedure. This is the same information provided by **prof**[DECd], the Unix profiling tool. Unlike **prof**, which uses periodic sampling of the program counter to estimate the amount of time spent in the body, PROTEUS' profiling information is exact.

Both PROTEUS and **prof** measure the time spent in the body of each procedure, which excludes the time spent inside called procedures. This metric is useful because it reveals which procedures are active the most; including the time of child procedures reveals which procedures are on the stack the longest. The latter is less useful since it is difficult to tell if the bulk of the time is spent in the body or in the called procedures. Furthermore, it is easy to measure the elapsed time spent in a procedure directly: record the entry time and subtract it from the exit time.

Using the profiling information, the **stats** program produces three graphs in table or bar-graph format: the number of calls for each procedure, the total number of cycles spent in the body of each procedure, and the average cycles per call for each procedure. To help reveal the most time-consuming procedures, **stats** lists the procedures in descending order of total time.

Augmenting a procedure with profiling code requires four steps. First, **augment** must allocate storage for the number of calls and the total time. Second, **augment** must add code to the prologue to increment the number of calls. Third, it must add code to each basic block to update the total-time counter. Finally, **augment** must communicate the location of the profiling information to the simulator engine.

To allocate storage, **augment** generates a data area consisting of two four-byte integers that are initialized to zero. If the procedure being augmented is **foo**, then the label for the data area is **foo_P**.[4] By convention, the first integer contains the total time and the second contains the number of calls.

Incrementing the number of calls is straightforward. In the prologue, after the code that checks for stack overflow, **augment** inserts instructions to load, increment, and store the number of calls, which is the second integer located at address **foo_P**.

The extra code to update the total time is also simple. For each basic block, the total-time

[4] If **foo_P** is used elsewhere, this will result in an error at link time. The only solution is to rename either the procedure or the other use of **foo_P**. Although this is ugly, it has never occurred in practice (to my knowledge).

counter is incremented by the number of cycles required for the block. This code is identical to the normal cycle-counting code except for the choice of counter. The cycle counter and the total-time counter are always updated at the same time and by the same amount, which ensures that the total-time counter correctly tracks the simulated time spent in the body of the procedure.

Finally, **augment** generates code to communicate the address of the profiling information to the simulator engine. In the code added to the prologue, the routine `InitProfile_` is called if the number of calls is zero. Thus, the first time the procedure executes the address of `foo_P` is passed to `InitProfile_`, which records the address for future use. At the end of the simulation, PROTEUS uses the recorded addresses to save the profiling information to the event file. Note that the initialization code runs at most once per procedure.

Unfortunately, the profiling code nearly doubles the overhead due to augmentation. However, as we shall see in Section 4.10, the overhead due to augmentation is not a significant factor in simulator performance.

4.7 Cycle-Counting Libraries

To allow user code to exploit the rich collection of C library routines, the simulator must provide cycle-counting versions of these routines. This requires both building the cycle-counting libraries, and correctly linking them in with the user code.

The simulator includes cycle-counting versions of the standard C library, `libc.a`, and the math library, `libm.a`. These were created by augmenting the source code for these libraries. The new versions are `libcyc.a` and `libcycm.a` respectively.

Correctly linking in the cycle-counting versions is a more difficult problem, because the simulator may need both the normal version and the cycle-counting version of the same routine. For example, **strlen**, the library routine that computes the length of a string, may be called both from the simulator engine and from an augmented user procedure. The engine requires the normal version, while the user code requires the cycle-counting version.

To resolve this problem, the two versions have distinct names. Cycle-counting versions are prefixed by the string "cyc_", while the normal versions have the normal name. The normal version of **strlen** is simply called "**strlen**", while the cycle-counting version is named

"cyc_strlen". This naming convention allows PROTEUS to link in both versions of a library routine.

Using separate names only partially solves the problem: users must remember to call the correct version. This is painful, since it is easy to forget the "cyc_" prefix, resulting in code that works correctly but ignores the time required by the library routine. Furthermore, not all library routines have a cycle-counting version, in which case prepending "cyc_" leads to an identifier that the linker cannot resolve. In general, the cycle-counting versions exist only if their costs are reasonable for a multiprocessor. For example, the file I/O routines do not have cycle-counting versions because it is unlikely that the costs for these routines on a multiprocessor are the same as on a Unix workstation.

To avoid the problems of two distinct names, **augment** automatically prepends "cyc_" to any reference to a library routine that has a cycle-counting version. This hides the fact that there are two names: references to library routines in cycle-counted code refer to the cycle-counting versions (when they exist), while references in non-cycle-counted code refer to the normal versions. The user need not worry about which version will be used, nor even know that there are two versions.[5] Finally, if the user references a library routine in augmented code that does not have a cycle-counting version, **augment** displays a warning that it is referencing the normal version,

4.8 Turning Counting On and Off

It is common in sequential programming to add debugging or monitoring code arbitrarily. In concurrent systems, however, the addition of such code may cause the monitored effect to disappear because of slight changes in timing, a problem called the *probe effect* [Gai86]. PROTEUS allows users to add debugging and monitoring code that is not cycle counted, and thus does not change the behavior of the system. (In rare cases, the additional code may affect the timing of the system; this is discussed below.)

To allow zero-cost monitoring code, PROTEUS provides macros to turn off cycle counting temporarily:

[5]The translation of library references is the only case where **augment** modifies the source code (as opposed to merely adding lines).

Macro	Function
CYCLE_COUNTING_OFF	Turn off cycle counting
CYCLE_COUNTING_ON	Turn on cycle counting

These macros should always be used in pairs. The code between the macros is not cycle counted, although direct changes to the cycle counter still occur. (Explicitly changing the cycle counter is discussed in Section 4.9.)

This functionality is implemented via volatile global variables named `cycles_on` and `cycles_off`. The macros expand into reads of the corresponding variables; because the variables are declared "volatile", the C compiler is not allowed to throw away the reads, even though the returned values are never used. During augmentation, **augment** detects references to these variables. When a reference to `cycles_off` is encountered, **augment** removes it and turns off cycle counting. Likewise, **augment** deletes references to `cycles_on` and reinstates cycle counting.

Although the code between the macros is not cycle counted, it may still affect the cost of the surrounding code indirectly, which could change the behavior of the system. For example, if the additional code uses most of the registers, the surrounding code may have to do more memory accesses than it did before. In this case, the "nonintrusive" code slightly increases the cost of the surrounding code.

We have *rarely* observed this problem in practice; the addition of monitoring code to cycle-counted code has not caused the effects being studied to disappear. Should it occur, however, it is possible to adjust the cost of the monitored code so that it matches the cost it had prior to the addition. PROTEUS provides primitives for increasing and decreasing the cycle counter by a delta, so it is easy to subtract out the extra cycles due to the monitoring code. The next section describes how to alter the cycle counter explicitly.

4.9 Explicit Control of The Cycle Counter

It is sometimes useful to alter the cycle counter directly. The main reason for altering the count is to vary the costs of a particular operation without changing its functionality. For instance, to simulate an architecture with a high cost for thread creation, one would simply alter the cycle counter directly in the thread creation routine. This allows direct control of costs without

affecting correctness or overall simulator performance.

Three macros are used to access the cycle counter; they are defined in `user.h`:

Macro	Function
`CURR_TIME`	Evaluates to the current time
`AddTime(t)`	Adds `t` cycles to the global time
`SubtractTime(t)`	Subtracts `t` cycles from the current time

To read the current time, use the macro `CURR_TIME`. It generates a special marker in the assembly language that causes `augment` to replace the marker with the value of the cycle counter.[6] The marker ends the current basic block so that the value read from the counter is exact. (If the marker did not end the block, the counter would contain the time of the last instruction in the block.)

The arguments to the `AddTime` and `SubtractTime` macros should be small integers. To add a large amount of time to the cycle counter, repeatedly add a small number by placing the `AddTime(t)` statement in a loop. Using this technique, the quantum check is performed each time through the loop, which increases the accuracy of the simulation.

As described in the last section, there are cases where monitoring code in a non-cycle-counted section of a cycle-counted file indirectly affects the cost of the surrounding code. In such cases, the cycle counter should be altered directly to remove the indirect costs of the monitoring code. The most successful way to remove these effects is to look at the assembly language with and without the monitored code. The basic block structure will be the same except for the additional code, thus one can measure the difference and subtract it out.[7] A simpler but less reliable method is to measure the time of the surrounding code with and without the monitoring code, and then to subtract the difference. It is probably sufficient to guess a small number of cycles, since the indirect effects tend to cause very minor increases in the cost of the surrounding code.

To improve performance, the cycle counter actually counts *down* from the quantum to zero. When the counter becomes negative, the quantum has expired. There are three global variables

[6] This macro has two definitions: one for cycle-counted code and a second for non-cycle-counted code. The one described here is for cycle-counted code; the other definition returns the value of the cycle counter directly, and is discussed below.

[7] The `SubtractTime` macro must be placed inside the non-cycle-counted area; otherwise *it* will change the timing.

involved in cycle counting: `cycles_`, `quantum_`, and `base_time_`.[8] The `cycles_` variable counts down from `quantum_`, and the current time is computed by:

```
current_time = base_time_ + quantum_ - cycles_
```

(This is the definition of the `CURR_TIME` macro when it is used in non-cycle-counted code.) Thus, the `AddTime(10)` macro actually *subtracts* ten from `cycles_`. Users should use the macros instead of altering `cycles_` directly, and should never alter the `base_time_` and `quantum_` variables.

4.10 Optimizations

There are quite a few optimizations that could be made to **augment**. Compacting the basic-block graphs is one that is implemented, as is the use of register `$1` when the general registers are in use. Before looking at other optimizations, it is important to evaluate what effect they may have.

Surprisingly, optimizations to **augment** have almost no effect on simulator performance. This is because the amount of time spent in augmented code is a small fraction of the total execution time. Figure 4-5 presents a procedure-level profile of the simulator, which reveals that the eight-queens problem spends only 1.4 percent of its execution time in augmented code. Since placing a queen is pretty simple, this number may be lower than for other applications; however, the order of magnitude is representative. The key point is that the overhead from simulating the network and the shared-memory system and the overhead from context switching dominate the overhead from augmentation. Thus, an optimization that halves the overhead of augmentation reduces simulator execution time by only a few tenths of one percent.

Because of the meager rewards, few optimizations to **augment** have been implemented. For completeness, this section discusses two possible optimizations. The first optimization keeps the value of `cycles_` in a register across basic blocks, instead of loading and storing the global in every block. The value would be loaded only in the prologue and after procedure calls, and stored only in the epilogue and before procedure calls. A register other than register `$1` must

[8] These variables are declared as external unsigned long integers.

Percent	Time (seconds)	Procedure (file)
13.3	10.2200	**net_delay** (net.c)
11.5	8.9000	**ctxsw** (ctxsw.s)
5.9	4.5500	**memory_controller** (shmem.c)
5.9	4.5400	**next_request** (req_queue.c)
4.9	3.7800	**Resume_Handler** (resume.c)
4.7	3.6000	**main** (main.c)
4.1	3.1600	**build_new_packet** (net.c)
3.9	2.9700	**send_protocol_message** (shmem.c)
3.7	2.8900	**Send_Packet_Handler** (net.c)
3.6	2.7800	**CheckForInterrupts** (ihandler.c)
3.3	2.5500	**Route_Packet_Handler** (net.c)
2.7	2.1000	**TrapToSimulator** (resume.c)
2.4	1.8700	**SimCall** (simreq.c)
2.3	1.7900	**Nocache_Pkt** (shmem.c)
2.3	1.7500	**net_hops** (net.c)
2.2	1.6800	**new_request** (req_queue.c)
2.1	1.6500	**enqueue_request** (req_queue.c)
2.0	1.5700	**dispatch_to_controller** (net.c)
1.7	1.3300	**newpkt** (net.c)
1.7	1.3300	**shared_memory** (shmem.c)
1.7	1.2800	**Shared_Memory_Write** (shmem.c)
1.3	1.0100	**processor_request_nocache** (shmem.c)
1.3	1.0000	**ReadTimer** (timer.c)
1.2	**0.9100**	**add_new (queens.ca)**
1.2	0.9100	**Shared_Memory_Read** (shmem.c)
...	...	...
0.2	**0.1300**	**new_partial_solutions (queens.ca)**
0.0	**0.0300**	**stage (queens.ca)**
0.0	**0.0200**	**GETMEM (queens.ca)**

Figure 4-5: Procedure-level profile of PROTEUS running eight-queens. Only the file "queens.ca", highlighted in **bold**, is augmented. The augmented code accounts for only 1.4 percent of the total execution time. In contrast, simulating the network (**net_delay**) and context switching (**ctxsw**) account for nearly one-quarter of the execution time.

be available to hold the value, since the assembler will need register $1 for user code. The interaction of cycles_ with user access to the current time makes this tricky to implement, although not impossible. This optimization also has the disadvantage of making it difficult to compute the current time when debugging, since the value of the global may be stale. These problems, combined with the optimization's minor effect on performance, make this optimization not worth implementing.

A second optimization would dedicate a register to hold the value of cycles. This would eliminate all of the loads and stores to cycles_ in the user code. Unfortunately, this is difficult to implement without modifying the C compiler.[9] The compiler may use all of the available registers, which would require augment to switch back to loading and storing the cycle count. This complexity of this optimization and its small effect on overall performance make it not worth implementing.

In general, optimizations have been implemented only when they are simple and clearly safe. In practice, this leads to optimizations whose effects are clearly local to the point of augmentation. For example, the use of register $1 for temporary storage, instead of spilling a register to the stack, only affects the cycle-counting code of that particular block. The .noat and .at directives guarantee that augment and the assembler do not interfere in their use of the register. The simplicity and safety of this optimization made it worth implementing, even though the resulting performance improvement is quite small.

4.11 Conclusion

This chapter discussed the design of augment in detail. It presented the code generated for each transformation, and discussed the major problems that the design had to resolve. Given the detailed design of the augmentation process, the next chapter looks at some experiments to empirically evaluate the overhead and correctness of augment.

[9] Unlike the MIPS compiler, the Gnu compiler, gcc, allows users to mark particular registers "off limits" to the compiler, which makes it easy to dedicate a register to tracking the current time.

Chapter 5

Experiments

This chapter evaluates the overhead and correctness of **augment** by discussing the results of three experiments. The first experiment investigates the overhead of **augment**, the second verifies the accuracy of the cycle counts, and the third provides evidence to validate the simulator as a whole.

These experiments use **pixie** [DECc], a production-quality cycle-counting utility originally developed by MIPS and included with versions of Unix for MIPS processors. The **pixie** utility is very similar to **augment**, except that it is intended only for MIPS object code files. Instead of augmenting assembly language files, **pixie** augments binary executables. Thus, it first disassembles the machine code into assembly language. Once the input is in assembly language, the utility locates basic blocks and augments each one. Like **augment**, **pixie** assumes that the cache hit rate is 100% and that instructions take a fixed number of cycles. However, **pixie** is more accurate than **augment**, since the MIPS assembler inserts pipeline delays and expands macros before generating the machine code. Thus the code input to **pixie** matches the machine code exactly, while the code input to **augment** is merely a close approximation. Since **pixie** has been in use for several years, it is reasonable to assume that it generally produces the correct cycle counts.

Program	Normal Cycles (time)	Augmented Cycles (time)	Overhead Factor
Queue	65,876,631 (3.1)	144,581,647 (6.6)	2.2
Sieve	52,483,384 (2.1)	130,868,590 (5.2)	2.5
augment	11,670,316 (1.0)	24,578,648 (1.8)	2.1

Table 5.1: Measuring the overhead of **augment**. This table compares several programs with and without augmentation. The cycles were determined by **pixie**, and the time in seconds by the Unix **time** command. The overhead factor is the ratio of the **pixie** cycle count for the augmented version over that of the normal version. The overhead is consistently a small factor.

5.1 Measuring the Overhead

This section measures the overhead of augmentation for three C programs. The execution times for each program are measured with and without augmentation. The ratio of these times is the overhead factor. The three programs are **Queue**, **Sieve**, and **augment**.

Queue performs 1.2 million operations on a priority queue. It is one of several priority queue implementations that were considered for an important data structure in the simulator engine. The selection of operations was generated by tracing the execution of the eight-queens program on the simulator. (The algorithm is essentially a calendar queue [Bro88].)

Sieve computes the first 100 prime numbers, starting over for each prime. To determine primality, **Sieve** divides the potential prime by every smaller prime until it either finds a factor or exhausts the smaller primes.

The third program is **augment** itself, which executes some routines that are not cycle counted. Since the overhead for non-cycle-counted procedures is zero, the overall overhead factor should be slightly smaller than in the other programs.

Table 5.1 show the results of these measurements. Each run is timed in two ways: first, using the Unix **time** command [DECa], and second, using **pixie** to count the total number of cycles executed. The **pixie** number is far more accurate: the **time** results are affected by other processes. The overhead factor is the ratio of the cycle count for the augmented version over the cycle count for the normal version.

The overhead is consistently a small factor. Using **augment** as the input file, which includes

Program	pixie Count	augment Count	Percent Error
Queue (MIPS)	65,876,631	65,931,364	0.083%
Queue (Gnu)	87,198,232	87,956,339	0.86%
Sieve	52,483,384	52,768,111	0.54%
augment	11,670,316	11,169,127	-4.3%

Table 5.2: Verifying the cycle counting of **augment**. This table shows that most of the time the cycle counts produced by **augment** are extremely accurate. The inaccuracies are discussed in Section 5.2.

some non-cycle-counted code, produces the lowest overhead as expected. Its overhead is about 5% lower than the input file with the second lowest overhead.[1] As we shall see in the next experiment, the non-cycle-counted code executes about 5% of the time. The overhead of the **Sieve** program is probably higher because of a low average basic-block size; small blocks incur a higher percentage of overhead.

As discussed in Section 4.10, the overhead of **augment** is less than 2% of the total overhead of PROTEUS: reducing the overhead of **augment** would affect simulation times by a very small amount.

5.2 Verifying the Cycle Counts

The second experiment evaluates the accuracy of the cycle counting by comparing the results of **augment** with the results of **pixie**. The latter should be slightly more accurate, since it augments machine code while **augment** works with assembly language. The same three programs are examined as in the first experiment. However, the **Queue** program is evaluated with both the MIPS C compiler and the Gnu C compiler. The data for the second compiler reveals that changes in the compiler have a much greater impact on the cycle count than the inaccuracies of **augment**. The results are summarized in Table 5.2.

These results show that with one exception, the counts produced by **augment** closely match

[1]The 5% number is the difference between the overhead factors for **augment** and **Queue**, which is .1, divided by the overhead factor for **augment**, which is 2.1.

the counts produced by **pixie**. The result for input file **augment** is the exception. The 4% difference is due to the existence of non-cycle-counted code. This code costs zero cycles according to **augment**, but costs the correct amount when measured by **pixie**. The inaccuracies of the other input files are due to the fact that **augment** counts assembly language instructions, not machine language instructions (as **pixie** does). As discussed in Section 4.3, the MIPS assembly language does not have a one-to-one correspondence with the machine code. The assembler uses context-specific macro expansion and inserts null instructions as required by pipeline delays. The unpredictability of the macro expansion and the extra cycles due to pipeline delays create discrepancies between the cycles required by the assembly language and the cycles required by the machine code. These discrepancies account for the slight inaccuracy of **augment** when compared to **pixie**.

These inaccuracies appear even more insignificant when we examine the effect of the compiler on the cycle counts. Changing the compiler for the **Queue** program results in a 31% change in the cycle count. This difference far exceeds the change due to inaccuracies in **augment**. The error due to **augment** is irrelevant given that the cost of a local instruction block can vary by 30% just because of the compiler.

5.3 Validating the Simulator

This experiment, originally performed by Chris Dellarocas, seeks to answer a very important question: given all of our assumptions, can we trust the results of a simulation? This section compares simulation results with published results from a real multiprocessor. If the simulator produces valid data, then its results should match those of the real multiprocessor, at least in their overall trends and features.

The experiment is a comparison of spinlock implementations by John M. Mellor-Crummey and Michael L. Scott from their paper "Synchronization Without Contention" in the proceedings from the 1991 *International Conference on Architectural Support for Programming Languages and Operating Systems*. Their code was modified to run on PROTEUS in less than one day.

Figure 5-2 reports the results of an experiment performed on a Sequent Symmetry multiprocessor. We ran the same experiment on PROTEUS: our results appear in Figure 5-1. The

vertical axis is the average time spent trying to acquire a spinlock. The horizontal axis represents the number of processors trying to acquire the spinlock. The major conclusion drawn from Figure 5-2 is that the simple test-and-test-and-set spinlock does not scale well; the other implementations do scale well. The PROTEUS graph leads to the same conclusion.

There are two differences between the graphs that deserve further discussion. First, the Sequent graph has a large dip for the test-and-test-and-set spinlock that is absent from the simulation graph. The dip is extremely counterintuitive: it implies that as the number of processors trying to acquire the lock *increases* from 14 to 16, the average spinning time *decreases*. In other words, if two of the sixteen processors stopped trying to acquire the lock, the waiting time for the rest would increase. Mellor-Crummey and Scott were unable to explain the dip and agree that it does not make any sense.[2] It is unlikely that the dip would appear in other experiments or on other bus-based machines. The data from PROTEUS is probably more representative of spinlock behavior on bus-based multiprocessors. If we discard the data for 15 and 16 processors, which effectively removes the dip, the slopes of the graphs match extremely well.

The second difference is the slight reordering of the other three spinlocks. In Figure 5-2, the Anderson lock performs slightly better than the test-and-set with exponential backoff. In the PROTEUS graph, the order is reversed. However, in both graphs these spinlocks are very competitive; that is, the points are close together. Since we used a generic bus simulation instead of a detailed simulation of the fairly complex Sequent bus, we expect some error in the absolute numbers. Furthermore, the effects of the compiler can move individual points ten to thirty percent. Given that we expect some error in the individual points, we could not draw any conclusions about the relative merits of the three implementations that scale well.

Overall the graphs match quite well. The perturbations seen with small numbers of processors match, as do the slopes for each implementation. Conclusions drawn from the PROTEUS graph are valid for the Sequent, which validates the accuracy of PROTEUS in this case.

Finally, evidence for the accuracy of PROTEUS comes from other sources as well. We have also reproduced published results for sorting algorithms on an nCUBE [FJL+88, Qui89]. In his research on concurrent search trees, Adrian Colbrook found that PROTEUS reproduced his

[2]This statement is paraphrased from a private communication from John Mellor-Crummey.

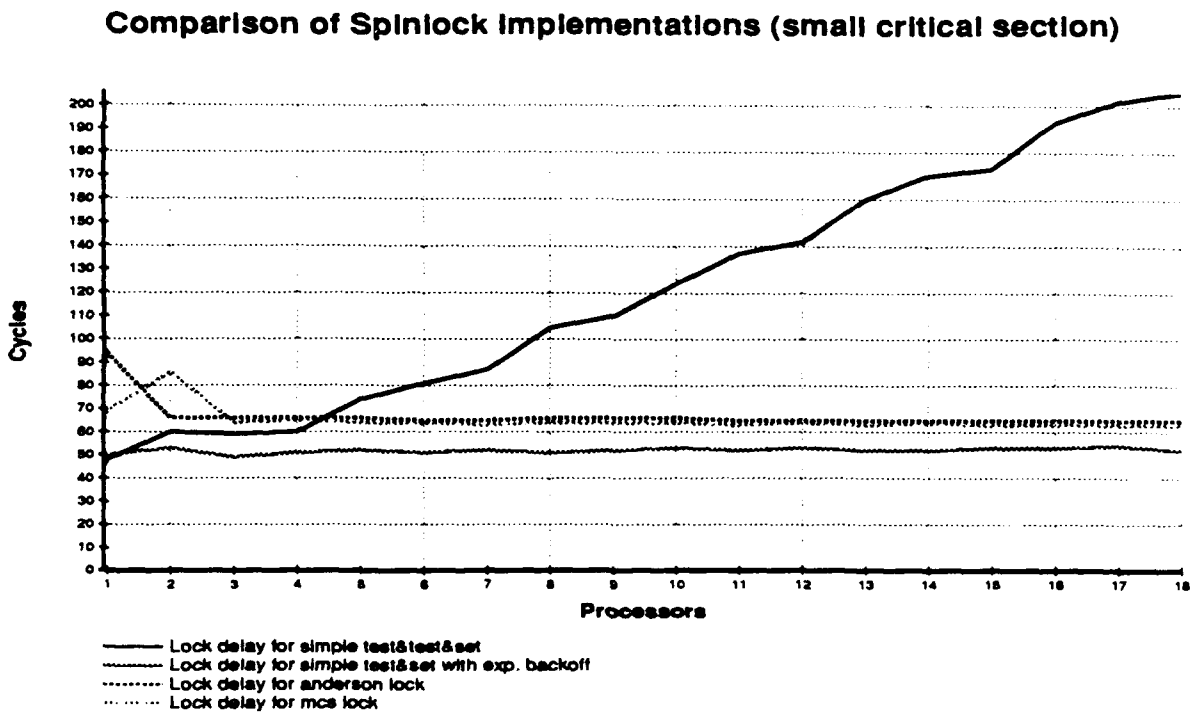

Figure 5-1: Spinlocks with a small critical section simulated by PROTEUS.

data from a Supernode multiprocessor quite well [CBDW91]. Thrashing problems predicted by David Chaiken for his coherent-cache protocol were produced in PROTEUS simulations [Cha90]. In general, any effect that we expected to see has actually appeared.

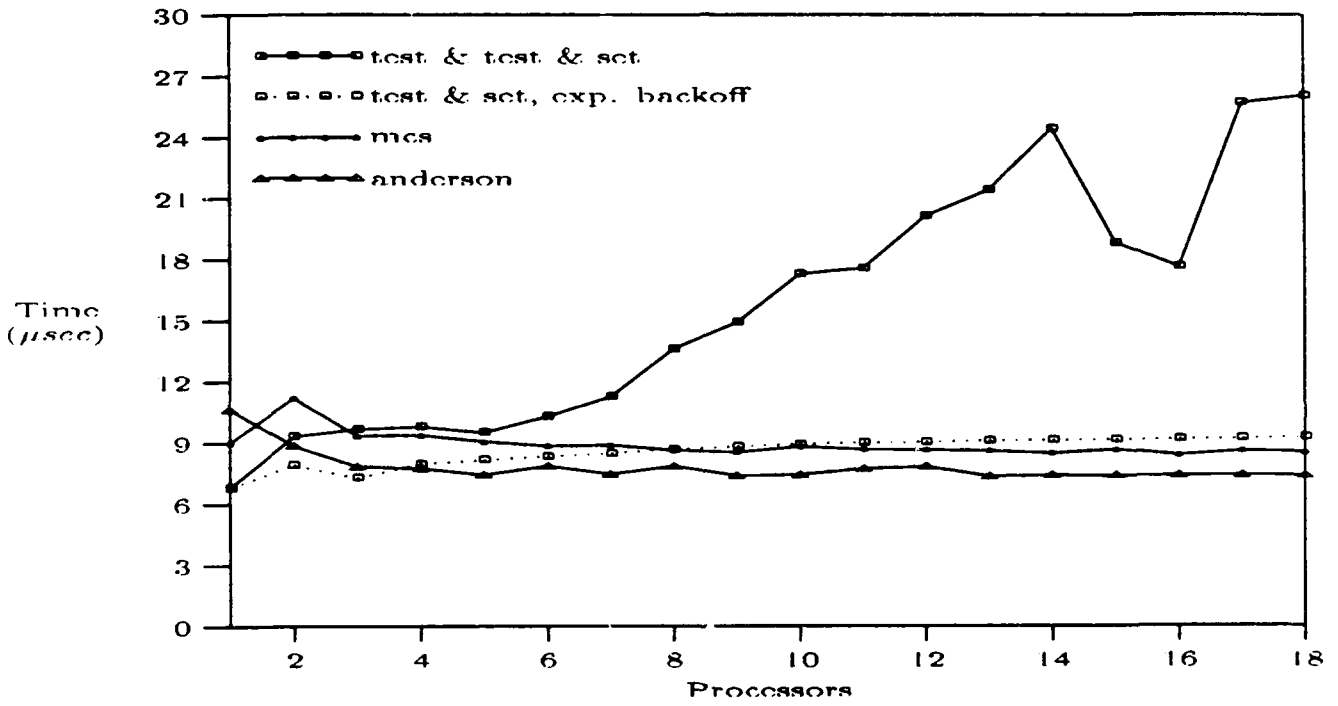

Figure 5-2. Same comparison on the Sequent Symmetry, from Mellor-Crummey and Scott [MCS91].

Chapter 6

Related Work

This chapter discusses several multiprocessor simulators that use direct execution as well as some related uses of augmentation. There are several simulators that use some form of direct execution. For the purposes of this thesis, the primary difference in these systems is how they account for the time required by the directly executed code.

6.1 Aspen

The *Aspen* simulator, the doctoral project of I. J. P. Elshoff at the University of Arizona, executes code directly and uses Unix timing routines to time the execution [Els91]. This method has the advantages of portability and improved performance over simulators that interpret instructions. The primary problem with this approach is the millisecond accuracy of the Unix timer. All times are implicitly truncated to the nearest millisecond, roughly ten thousand instructions for a ten MIPS machine. This is far too coarse for simulating a multiprocessor. In ten thousand cycles, a single processor in the J-machine could send a hundred messages [D+89].

A second serious problem is that it is not possible to add nonintrusive monitoring or debugging code, since all of the code is timed. The accuracy of the timer exacerbates the problem because it prevents users from subtracting out the cost of the monitoring code.

6.2 CARE

The CARE simulator developed at Stanford by Delagi, Saraiya, Nishimura, and Byrd, simulates LISP code by direct execution [DSNB87]. It uses a microsecond timer to track processor times. This is a viable approach if a microsecond timer is available to the simulator. However, it is not as accurate as counting instructions for processors that execute more than a million instructions per second. Even with a microsecond timer, the CARE simulator would not meet the goals of our simulator for two reasons. First, the performance is relatively poor. A shared-memory access requires on the order of ten to twenty milliseconds to simulate, while the same access in our simulator requires less than a thousand instructions, or about 30 microseconds on a DECstation 5000.[1] This discrepancy arises primarily from the overhead introduced by several layers of abstraction between the LISP code and the machine language.

A second deficiency results because many language design decisions involve the low-level code of the run-time system. The simulator must be able to measure run-time system costs accurately. Using LISP for the run-time code is unacceptable because there are many hidden costs in the high-level semantics of LISP. These hidden costs are counted by the CARE simulator even though they would not exist in the actual implementation of the run-time system. (These hidden costs also account for part of the performance degradation.)

Like Aspen, CARE prevents the use of nonintrusive monitoring or debugging code. With the improved resolution of the timer, users can attempt to subtract out the cost of monitoring code, but this is off by roughly half of a microsecond per instance.

6.3 Accounting for Time by Hand

The multiprocessor simulator developed by Luigi Rizzo, at the Universitá di Pisa, requires users to specify the costs of operations using a procedure call [Riz89]. A block that costs ten units must include a statement of the form `takes(10)`, which simply adds ten to the simulated time for this processor. These calls can be inserted anywhere; placing one in each basic block is essentially augmentation by hand. Besides being tedious, the use of **takes** calls is generally

[1] The CARE figures are from a private correspondence from Bruce Delagi. Our simulator takes substantially less than one thousand cycles if the accessed value is in the simulated cache, perhaps only thirty cycles.

inaccurate. The user may be quite far off in his estimate of the time required for a particular block or procedure. Augmentation provides the same control of costs as these calls, but defaults to the cycles required by the block instead of defaulting to zero.

6.4 Threads

The *Threads* simulator [MF88], developed by Mathieson and Francis at La Trobe University in Australia, was the first to use augmentation to account for the costs of instructions. Instruction counting is performed by inserting a procedure call into each basic block to update the simulated time. Although their simulator exploited direct execution, it requires many questionable assumptions. For example, they assumed that all memory accesses required the same number of cycles, and that all instructions require one cycle to execute. The machine model supports communication only through shared memory; *Threads* cannot simulate distributed-memory machines or interprocessor interrupts.

6.5 RPPT

The Rice Parallel Processing Testbed (RPPT) [CMM+88], which was developed by Covington et al. at Rice University, is similar to PROTEUS but is oriented towards message-passing multiprocessors with tens of processors. It uses code augmentation to account for costs, but does not address issues such as stack overflow, profiling, and nonintrusive monitoring and debugging. The paper discusses future plans for validating RPPT, but does not present any results.

6.6 Tango

The Tango system [DGH90], developed by Davis, Goldschmidt and Hennessy at Stanford, is the closest simulator to our work. Developed concurrently with PROTEUS, Tango uses augmentation to determine the time required by local instructions. The augmentation overhead is nearly the same as in PROTEUS. The overall system performance, however, is one to two orders of magnitude worse than PROTEUS. The primary reason for this difference is *Tango*'s use of one Unix process for each thread, which results in a typical context-switch time of 180–250

microseconds (according to the authors). The lightweight threads package developed for PROTEUS, combined with invariants that reduce the amount of state that must be swapped, results in an average context-switch time of about five microseconds. Tango uses Unix semaphores for synchronization, which have higher overhead than the semaphores developed for PROTEUS. The simulation overhead of synchronization in PROTEUS is comparable to a procedure call, as opposed to the cross-process communication required by Tango. Furthermore, Tango requires that all shared-memory accesses call the memory subsystem, which executes in a different process; each simulated access requires two context switches and takes on the order of one millisecond. In PROTEUS, a shared-memory access results in a context switch only if the value is not in the simulated cache. This brings the average access time down to a few microseconds.

6.7 Other Uses of Augmentation

Augmentation has many uses besides simulation via direct execution. The original use, presented by Weinberger, was counting instructions. The other major use is the addition of code that generates event traces dynamically.

6.7.1 Profiling

Augmenting assembly language with code that counts instructions was first documented by P. J. Weinberger at Bell Labs [Wei84]. His article *Cheap Dynamic Instruction Counting* covers the basics of augmentation quite well. The tool generates execution profiles of sequential programs by counting the number of times each (high-level language) statement executes. The counts are very accurate and thus give quite a good picture of where the program spends its time.

The MIPS corporation rediscovered augmentation for profiling and produced the **pixie** utility [DECc]. As discussed in the last chapter, **pixie** is a very accurate profiling tool that uses augmentation to count cycles dynamically. Its primary advantage over **augment** is its ability to augment machine code directly instead of using assembly language. Augmenting machine code is more accurate since the assembler performs macro expansion and inserts pipeline delays. Augmenting machine code requires neither recompilation nor access to source code. One advantage to augmenting assembly language, however, is that it is easy to augment only a subset of

the original code by using directives in the input file to turn augmentation on and off. A second advantage is ease of portability. Changing **augment** to handle a different assembly language requires only defining each instruction and its cost, and determining which instructions delimit basic blocks and procedures. Augmenting machine code requires knowledge about the object code format and about the specific processor. The disassembler requires detailed knowledge of the instruction set of the *exact* processor, rather than knowledge about the instruction set architecture provided by the assembler, which is less restrictive.

6.7.2 Tracing

Augmentation has also been used to produce address traces for both sequential and parallel shared-memory machines. Traces are useful as input to memory system simulations because they represent the access patterns of real applications. The basic idea is to augment a program so that it records every memory reference as it executes. This is done by inserting code at every reference to record the address. Trace generation through augmentation is described by Eggers et al. [EKKL90] and by Stunkel and Fuchs [SF89].

6.8 Conclusion

Our decision to use augmentation was influenced most by *Aspen*, which provides the performance of direct execution, but not the accuracy of augmentation. The use of augmentation in multiprocessor simulators was first used by Ian Mathieson and was rediscovered by ourselves and by Helen Davis at Stanford for the Tango simulator. Although Tango is very similar to PROTEUS, it was not an influence since it remained unpublished until PROTEUS was nearly complete. The idea to apply augmentation to a multiprocessor simulator resulted in part from the success of augmentation for generating traces, epitomized by the work of Eggers et al. [EKKL90] and Stunkel and Fuchs [SF89].

Chapter 7

Conclusions

This thesis investigated the combination of code augmentation and direct execution as a key facet of PROTEUS, a versatile high-performance parallel-architecture simulator. After providing an overview of PROTEUS, we discussed the need for both performance and accuracy in a multiprocessor simulator, and the fundamental conflict between them. This conflict leads to the principle of maximizing performance by providing only the required accuracy for a particular simulation.

After establishing the motivation for code augmentation and providing an overview, this thesis examined the detailed design and implementation of code augmentation. It then presented empirical evidence that supports the correctness and low overhead of the implementation, and compared PROTEUS results with those of a real multiprocessor to validate PROTEUS as a whole.

The use of direct execution and code augmentation to simulate local instructions is an application of the principle of maximizing performance by providing only the required accuracy. While instruction interpretation has an overhead of about two hundred instructions per simulated instruction, the overhead of augmentation is only about two instructions per simulated instruction. Thus, the simulation of local instructions via direct execution and code augmentation reduces the simulation overhead by about a factor of one hundred.

Furthermore, the thesis claims that the loss of accuracy in the move from instruction interpretation to code augmentation is unimportant for parallel-systems research. Intuitively, the intricate details of a particular instruction set are not relevant to the overall behavior of the system. The ability of PROTEUS to reproduce results from real multiprocessors is direct

evidence that code augmentation is sufficiently accurate.

The use of augmentation also allows precise control of the cost of a block of code. A block that is not cycle counted costs zero cycles, which allows nonintrusive monitoring and debugging code. Thus, users can add debugging code without worrying that the symptoms of the bug will disappear due to a reordering of nonlocal interactions. Furthermore, the cost of a block may be altered explicitly to produce more accurate costs relative to the target architecture. For example, the same message-passing code, with different costs, has been used to simulate message passing on several architectures.

We also use augmentation to collect profiling information. Because augmentation assigns the cost of each block, the profiling information is exact. In particular, non-cycle-counted code and code with explicitly altered costs are profiled correctly.

Augmentation aids debugging by adding code to detect stack overflow as soon as it occurs. The quick detection makes the error trivial to fix, which essentially eliminates a class of bugs that are notoriously difficult to track down.

The augmented code also limits the number of cycles that a thread can execute without returning control to the PROTEUS engine. This improves the accuracy of simulations, and allows the engine to remain in control even in the presence of infinite loops in user code.

Finally, one should note that augmentation fits in well with the rest of the compilation environment. Since augmentation is simply a transformation of assembly language, the rest of the tools for compilation and linking are inherited without modification from the host workstation.

PROTEUS as a whole meets all of our original design goals: it is fast, accurate, versatile, and provides substantial support for debugging, data collection and graphical output.

Thanks to augmentation and *fast context switching*, PROTEUS is consistently one to two orders of magnitude faster than comparable multiprocessor simulators. The range of performance and accuracy combinations provided by PROTEUS allows users to achieve the maximum performance for the amount of accuracy they require. This provides more than a factor of ten performance improvement during development, but allows slower more accurate simulations for data collection.

In its most accurate mode PROTEUS consistently produces accurate simulations. We have reproduced published empirical results from several multiprocessors and algorithms: the PRO-

TEUS graphs consistently match the data from real machines. In general, any effect that we have expected to see actually has appeared in simulation results. More importantly, all unexpected effects turned out to be valid data rather than a manifestation of inaccuracies inherent in PROTEUS.

The simulator has also met the goal of versatility. PROTEUS simulates both message-passing and shared-memory machines. Topologies include buses, k-ary n-cubes, and multi-stage indirect networks. The memory module simulates cache-coherence algorithms, full/empty bits, and atomic memory operations. The configuration program allows users to control nearly every facet of the target architecture.

Support for debugging includes a built in "snapshot" mode that allows users to examine the status of processors, threads, locks, memory, and the network. There is also support for tracking down memory-overwrite bugs, which would be exceptionally tricky to resolve without the tools provided by PROTEUS. The early stack overflow detection provided by augmentation is one such tool. Debugging PROTEUS applications is simplified further because users can exploit the power of standard sequential debuggers to complement the tools provided by PROTEUS.

The simulator also provides very general data collection and display facilities. In addition to a broad range of data collected by the engine, users can define their own data types to collect exactly the data they need. A sophisticated graphing program combined with a simple but powerful graph-specification language allows users to examine their data in a variety of useful ways, and produces camera-ready hardcopy of the graphs for publication.

Although we designed PROTEUS initially as a testbed for experimenting with language and runtime-system mechanisms, it has become clear through use that simulators like PROTEUS are also effective for developing parallel applications. The monitoring capabilities provided by PROTEUS make debugging and initial performance tuning significantly easier than on real machines. Using PROTEUS for development reduces the time required on real multiprocessors, which are expensive and scarce. Finally, PROTEUS allows applications to be developed for a wide variety of machines, including machines that only exist on paper.

Finally, PROTEUS as a whole is *effective in practice* as a tool for parallel-systems research. It has been very successful as the base for the design and implementation of a concurrent object-oriented language, PRELUDE [WBC+91]. It has also been used for extensive studies of fault

tolerance, including a set of over two thousand simulations run in just a few days. Finally, to date PROTEUS has generated the empirical results used in two parallel-systems research papers, one on concurrent search-tree algorithms [CBDW91] and another on algorithms for readers-writers locks [HW92]. The usefulness of PROTEUS is a direct consequence of the performance and accuracy provided by augmentation.

Bibliography

[BD91] E. A. Brewer and C. N. Dellarocas. PROTEUS *User Documentation.* Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, November 1991.

[BDCW91] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[BDCW92] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 Conference*, June 1992.

[Bro88] Randy Brown. Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10), October 1988.

[CBDW91] A. Colbrook, E. A. Brewer, C. N. Dellarocas, and W. E. Weihl. An algorithm for concurrent search trees. In *Proceedings of the 1991 International Conference on Parallel Processing (ICPP '91)*, pages III138–III141, August 1991.

[Cha90] D. Chaiken. Cache coherence protocols for large-scale multiprocessors. Technical Report MIT/LCS/TR-489, MIT Laboratory for Computer Science, September 1990.

[CLN90] D. Chaiken, B.-H. Lim, and D. Nussbaum. ASIM Users Manual. ALEWIFE SYSTEMS MEMO #13, August 1990.

[CMM+88] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1988.

[D+89] W. J. Dally et al. The J-machine: A fine-grain concurrent computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[DECa] Digital Equipment Corporation. *csh(1).* Ultrix 4.0 General Information, Vol. 3A (Commands(1): A-L).

[DECb] Digital Equipment Corporation. *lex(1).* Ultrix 4.0 General Information, Vol. 3A (Commands(1): A-L).

[DECc] Digital Equipment Corporation. *pixie(1).* Ultrix 4.0 General Information, Vol. 3B (Commands(1): M-Z).

[DECd] Digital Equipment Corporation. *prof(1).* Ultrix 4.0 General Information, Vol. 3B (Commands(1): M-Z).

[Del91] C. N. Dellarocas. A high-performance retargetable simulator for parallel architectures. Technical Report MIT/LCS/TR-505 (Master's Thesis), Massachusetts Institute of Technology, June 1991.

[DGH90] H. Davis, S. R. Goldschmidt, and J. Hennessy. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Computer Systems Laboratory, Stanford University, July 1990.

[DSNB87] B. A. Delagi, N. Saraiya, S. Nishimura, and G. Byrd. An instrumented architectural simulation system. Technical Report KSL 86-36, Knowledge Systems Laboratory, Stanford University, January 1987.

[EKKL90] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of ACM Sigmetrics 1990*, pages 37–47, May 1990.

[Els91] I. J. P. Elshoff. *Aspen.* PhD thesis, University of Arizona, (expected) June 1991.

[FJL+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volume 1: General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[Gai86] J. Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.

[GKM82] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.

[Goo83] J. R. Goodman. Using cache-memory to reduce processor-memory traffic. In *Proceedings of the 10^{th} International Symposium on Computer Architecture*, pages 124–131, June 1983.

[HW91] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. Technical Report MIT/LCS/TR-521, MIT Laboratory for Computer Science, November 1991.

[HW92] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings of the 1992 International Parallel-Processing Symposium*, March 1992.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 269–278. April 1991.

[MF88] I. Mathieson and R. Francis. A dynamic-trace-driven simulator for evaluating parallelism. In *Proceedings of 21st Hawaii International Conference on System Sciences*. volume 1 (Architecture), pages 158–166, January 1988.

[Qui89] M. J. Quinn. Analysis and benchmarking of two parallel sorting algorithms: Hyperquicksort and quickmerge. *BIT*, 29(2):239–250, 1989.

[Riz89] L. Rizzo. Simulation and performance evaluation of parallel software on multiprocessor systems. *Microprocessors and Microsystems*, 13(1):39–46, January/February 1989.

[SF89] C. B. Stunkel and W. K. Fuchs. TRAPEDS: Producing traces for multicomputers via execution driven simulation. In *Proceedings of ACM Sigmetrics 1989*, pages 70–78, May 1989.

[WBC+91] W. E. Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh. Anthony Joseph, Carl Waldspurger, and Paul Wang. PRELUDE: A system for portable parallel software. Technical Report MIT/LCS/TR-519, Massachusetts Institute of Technology, October 1991.

[Wei84] P. J. Weinberger. Cheap dynamic instruction counting. *AT&T Bell Laboratories Technical Journal*, 63(8):1815–1826, October 1984.

OFFICIAL DISTRIBUTION LIST

DIRECTOR — 2 copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Boulevard
Arlington, VA 22209

OFFICE OF NAVAL RESEARCH — 2 copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. Gary Koop, Code 433

DIRECTOR, CODE 2627 — 6 copies
Naval Research Laboratory
Washington, DC 20375

DEFENSE TECHNICAL INFORMATION CENTER — 2 copies
Cameron Station
Alexandria, VA 22314

NATIONAL SCIENCE FOUNDATION — 2 copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

HEAD, CODE 38 — 1 copy
Research Department
Naval Weapons Center
China Lake, CA 93555